Embodying Language Models with VLM-style Fusion and Behavioural-Lingual Cloning

Gabriel Parl Paredes-Larson
MComp(Hons) Computer Science

2022-2023

Submitted by Gabriel Parl Paredes-Larson

Declaration
This Dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this Dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

There has been a shift in AI capabilities research to multimodal models. Recent papers have shown that fusing language models with vision encoders and training jointly on paired data (e.g. captioned images) is effective for making language models 'multimodal' and able to make effective use of visual input. Interest along this vein has grown towards entirely embodying language models into typically reinforcement learning environments to further extend their usefulness. This paper seeks to advance work in this early area of research.

We identify that VLM fusion architectures might provide an effective way to embody language models, specifically for increasing the expressiveness with which their output is converted to actions, noting that this approach has not been used yet likely due to a lack of paired multimodal data between reinforcement learning environments and language. We find that the recent Video-Pre-Training (VPT) paper provides a way to procure such a dataset.

We therefore design and propose a novel fusion method applicable to agents and language models, using novel mutual-conditioning fusion and a novel truncated/temporal cross attention mechanism which temporally aligns text and video using silence tokens; we suggest extending the established paradigm of behaviourally cloning human experts to include cloning their utterances; and provide code which builds on the VPT repository to enable further work.

We then collect the proposed dataset for the Minecraft environment; train the proposed architecture on this data using a pre-trained agent from the VPT paper and a pre-trained language model from the Transformer-XL paper; and attempt to measure how language impacts the agent with various metrics. Due to difficulty with the VPT repository and time constraints, we are unable to test the agent in the MineRL environment, and otherwise achieve inconclusive results on performance, leaving investigation to future work.

0.1 Acknowledgments

Harish Tayyar Madabushi for his support and encouragement with this project

Contents

Introduction

Recent advances in visual language models like Flamingo (Alayrac, Donahue and Luc, 2022) and PaLI (Chen, Wang and Changpinyo, 2022) have shown that the capabilities of language models can be brought over to the visual domain by fusing them with visual models and training on paired language-image data. This is a large step towards making language models more generally useful. The GPT-4 technical report (OpenAI, 2023) is an excellent demonstration of this, with the visual input allowing for higher performance on university/high-school level examinations which have a visual component to the problems. What is especially exciting about these demonstrations is that they bring some of the most useful capabilities of language models, namely learning how to perform a task from a few examples (few-shot learning), to the image domain, as was tested in the Flamingo paper.

We are now seeing, starting with works such as SayCan (Ahn, Brohan and Brown, 2022) and now Inner monologue (Huang et al., 2022) and PaLM-E (Driess et al., 2023), how the capabilities of language models can be further adapted to make them useful in typically reinforcement learning environments, 'embodying' them. Since language models are currently the most capable models for reasoning (Kıcıman et al., 2023), in-context learning, common sense and wide knowledge bases (Brown et al., 2020), they pose a promising building block to build agents which are able to solve problems across a wide variety of tasks, and we are now seeing capabilities research head in this direction. The motivation for this project largely comes from seeing such progress and looking where capabilities can be further improved. However, as they currently stand these models largely mix language model into the new RL modality not through the fusion-VLM method of high bandwidth neural connections, but by translating the environment into low bandwidth natural language, passing that to the language model, and then feeding its response in natural language into another agent which does actions in the environment. Taking inspiration from the aforementioned VLM architectures, we propose that this design might be improved by fusing the language and agent models through neural connections and training them to understand each other's modality using paired datasets. However, no such datasets are available.

The VPT paper was first brought to the attention of this research because it offers a way to make state of the art agents in difficult environments from observation of unstructured online datasets; it proposes a novel way to scrape and organise unstructured online data to create behavioural cloning datasets from only video observations. It further demonstrates the usefulness of this approach by using this method to create a behavioural cloning dataset for Minecraft. The resulting agents trained on this dataset

(and then finetuned with reinforcement learning) set a state of the art in Minecraft for the first time a diamond pickaxe is crafted, while also being unique in that the agent had access to unsimplified Minecraft controls, using keyboard and mouse, unlike the majority of previous works which use abstracted actions to reduce the action space and ease learning. This paper was key to this work for two reasons: 1. it gave the realisation that such performance could be gotten just from online observations, which makes much of the difficulty of reinforcement learning vanish (as shown in their paper, RL achieves little without this behavioural cloning pre-training), 2. that language could be scraped in tandem with this data, permitting the construction of the paired video-action-language dataset required for a VLM-like fusion between an agent and a language model.

This work aims to bring together some of these ideas about multimodality, dataset/-training pipelines and embodiment together to propose a novel architecture and training method for language model embodiment, implement it building from the provided VPT GitHub repository and pretrained weights[39], collect appropriate data using the VPT method, train the model, and see if it provides a potentially useful direction for future research towards embodying language models.

Literature, Technology and Data Survey

2.1 Related Work

Visual Language Models Multimodal visual-language models have recently achieved large performance gains in image understanding, as seen by recent architectures such as CLIP (Radford, A., et al, 2021), Flamingo, GIT (Jianfeng, Zhengyuan and Xiaowei, 2022) and PaLI, BEIT-3 (Wenhui, Hangbo and Li, 2022), VideoBERT (Sun, Myers and Vondric, 2019).

This not only greatly improves the usefulness of these language models in vision-language tasks, but also achieves high benchmark results in pure vision tasks such as zero-shot ImageNet (Deng et al., 2009) classification and other similar benchmarks, as tested in through multiple of the aforementioned papers.

These papers achieve impressive performance in the ability for language models to incorporate information from the visual modality demonstrates the feasibility and usefulness of combining modalities, as well as how efficiently it can be done, with simple architectures like GIT reaching SOTA in multiple benchmarks at the time of release despite only having 3 billion parameters compared to competing models like Flamingo with 80 billion.

The work of Flamingo especially provides an example of how general language models representations' are and how this permits multimodality by using adapter layers to feed a language model visual data and then train only the adapter layers and achieve VLM SOTA (at the time) while keeping the language model frozen. It is shown during evaluation that the model also effectively brings the few-shot learning capabilities of large language models to the image domain. The advances these papers make and their methods and experiments make the intuitive feasibility of multimodal models such as the one proposed in this project much greater and provide much confidence for this work.

Embodied Language Models The recent usage of pretrained language models in typically reinforcement learning domains is a specific extension of the above multimodal paradigm which is a great inspiration for this project. The SayCan architecture (Michael Ahn, Anthony Brohan, and Noah Brown 2022) demonstrates using pretrained language models' existing representations for predicting actions in a complex real world environment without further training the model. This finding was a key inspiration to this work as it demonstrates that large pretrained language models' language representations form a large enough knowledge base and can be used effectively enough that they can be used

to predict reasonable actions (described in language) in many situations without further training; and that it is possible to transfer these lingual actions into the real world (i.e. with their grounding "SayCan" mechanism).

The SayCan paper does this by training a second model to be able to carry out a set of 'skills', each with a matched input instruction that triggers the execution of the appropriate skill. The different skills are labelled with appropriate descriptions, and the language model can then be use to evaluate which action description is most likely given a language instruction from the user. Further hints of the performance benefits of this multimodal LM + agent are made by Reid, G. et al 2022 by the improved performance they find in initialising agents using BERT weights, and the benefits of even plain language prompting in RL is shown by Stepputtis, C., 2020. The papers above made the idea of fusing a full language model into a reinforcement learning agent become the key interest of this project and, with the recent VLM papers, give this the perspective of a way forward for multimodality, fusing the language, visual and action space modalities necessary for embodiment.

In more recent approaches for embodying language models, such as "Inner monologue" (Huang et al., 2022), "Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents" (Wang et al., 2023), use a similar mechanism to the SayCan paper to communicate from the environment to the LM and the LM to the environment, i.e. from the environment state we first produce a language description, pass that into the language model model, and then (effectively) take the language models textual output, and pass that onto the agent/environment. In this way, unlike with VLM fusions, this relies on other models to translate between the modality to text, rather than getting the language model to learn itself how to do this translation.

The recent Palm-E paper is a step closer to the VLM method, as it passes environment state and vision embeddings for tokens into the language models - which does learn these additional modalities - rather than translating this data into language data before handing it to the LM. However, the output remains in language space only.

A key feature of the research mentioned, and to the knowledge of the researcher of the research attempted so far, these have a distinct difference from the VLM approach, which is that rather while VLMs allow information to be shared across different modalities through high bandwidth neural means like cross attention, current LM embodiments translate inputs (except PaLM-e) and outputs to and from low bandwidth language when passing information to/from the language model. They also do not train the different modality models to align their representations with paired multimodal data as VLMs do, but often use prompts engineered to make the language models' outputs relevant to the agent. The input side is addressed more recently in the PaLM-e paper by concatenating tokens of new modalities (in more detail later), but the output side has not yet been addressed, likely due to a lack of data linking language and actions. Without this data, the language model can only be trained to output actions, as the paper 'can Wikipedia help Offline Reinforcement Learning' (Reid, Gu and Yamada, 2022) does, by initializing an RL agent with the weights of the BERT (Devlin, Chang and Lee, 2019) language model. While this does improve performance, the model catastrophically forgets all language, removing important skills that we would like to preserve.

The VPT paper does an simple experiment with language where they condition the agent on the a language embedding of the most recent 30-second chunk of speech from

the user (the input is split, words to not come in at the spoken rate), and are able to use this to instruct the agent in language. While this is a step toward using language models to control such a model, this kind of language model is not able to to do the causal reasoning and in-context learning that would likely most benefit the agent, so it is not the kind of language model keyly being referred to by this work. The also mention that the steerability of this agent is 'weak'. It also cannot be used autoregressively, and so is unable to prompt itself during inference in the environment to do chain of thought prompting or explain what it is doing. It seems important to extend this work with a autoregessive language model. Indeed, the VPT paper notes how future research might look into predicting the next token in a similar setup. While this language experiment in the VPT paper was not in mind during the initial conceptualisation of this work, we nonetheless do propose a specific implementation for such an idea, extending the research in this suggested direction.

Language Model Advances While this is already somewhat covered in the above points, it is stated explicitly here that the recent and impressive advancements in language modelling performance by language transformers (Tom, B. B., et al, 2020) (Wei, J., 2022) are core inspiration for this work. Their demonstrations of capabilities such as few-shot learning, causal reasoning and wide knowledge bases have inspired this work to take advantage of them and put them to more effective use than in the pure language domain.

This work therefore aims to advance the expressiveness of language model embodiments using learnings from PaLI/Flamingo-style VLM model fusion, the VPT paper's methodology for data collection and multiple other architectural/data mechanisms (papers related to Transformer-XL and the use of silence tokens are brought up in the methodology, since they are more relevant to the precise implementation rather than background information).

2.1.1 Research Questions

We primarily look into the possibility bringing VLM-style fusion architectures for embodying language models, in order to increase the expressiveness with which language models can output actions. Secondly, we implement the architecture, the data collection pipeline and the training pipeline to see if this methodology is feasible in construction. Lastly we train the model with a (relatively) small amount of data and investigate if this architecture brings any quantitative or qualitative advantages - we test for a number of these described in R3 and R4.

R1. Can we construct an architecture which allow the modalities to be mixed such that the language model produces relevant text for the agent and the agent can condition its actions on its output? Can it be made to be differentiable, trainable and efficient?

R2. can we construct a pipeline to gather the required dataset? This will require a way to efficiently collect, label, clean and preprocess data. We will need to create code to give videos action labels using the IDM model the VPT paper provides. In order to do language modelling across time we will also need to transcribe the audio with token-level timestamps and find some method to align the tokens to the frames despite the

difference frequencies of occurrence. We investigate feasibility of producing an efficient data collection pipeline with the above features.

R3. It is hoped that the language model also allows the agent to be strongly steerable with manually inputted language conditioning during inference in the environment. This can be tested in two ways: 1. we can test how the loss on the behavioural cloning dataset is affected by obfuscating the language input and seeing if that affects the accuracy of the predicted actions. 2. testing in the environment: providing the agent with a prompt and measuring how different prompts affect its behaviour, like is done in the VPT paper e.g. if we prompt it with 'I am going to explore/run', does it go significantly further than it would otherwise? While we unfortunately cannot test in the environment, the first option should be equivalent to testing if the agent's actions causally depend on the given language signal. Finding this kind of causal dependence on language might enable longer term planning and few shot learning as the Flamingo paper found, so these should be tested for as well.

R4. It is further hoped that the language model outputs human-readable language that correlates with the agent's actions as the agent navigates the environment, providing some interpretability of the agent's actions.

Background

The Pretraining Paradigm Modern language models have introduced the paradigm of pre-train and finetune, which much research now uses (Devlin, J., 2019). This is because pre-training on a general enough dataset has been found to make models learn general and useful representations which allow for good performance on downstream tasks, and permit improved performance with small amounts of further finetuning. This paper draws from this paradigm by using two pre-trained models (a langauge model and an agent trained on Minecraft) so that it can minimise compute.

The Transformer Architecture (Vaswani et al., 2017) is a key component of this architecture; the agent and language model consist majorly of transformer blocks, and the cross attention layer used in this paper for fusing the models is also similar to a transformer block. The transformer has contributed to many state of the art results in deep learning in recent years across a range of modalities and tasks, such as the visual language models mentioned.

Large Unstructured Datasets Modern papers show that the use of large unstructured online datasets can result in useful generalisation abilities such as GPT and CLIP, which can achieve SOTA results on benchmarks even without being trained on any examples from those datasets. Especially motivating was the recent application of this paradigm to behavioural cloning data as shown my the VPT paper, mentioned earlier, resulting in large leaps in SOTA in Minecraft. These type of large performance improvements from data scrapes give the inspiration for continuing this direction of drawing behavioural cloning data from the internet, and inspires the decision to add language data from gameplay videos as a way of further taking advantage of available Big Data. While these ideas are mentioned elsewhere, it is important to separate out this paradigm and explicitly point it out.

Methodology

3.1 Architecture

The specific implementation of making three key adjustments to their behavioural cloning method with language conditioning from VPT's H.I experiment:

1. Replacing language conditioning with full language modelling of the experts' utterances during their navigation of the environment.

2. Making use of a pre-trained language model for language cloning, taking inspiration from existing multimodal approaches to fuse the agent and the LM, while proposing a way to extend this to mutual conditioning so each model conditions on information from the other modality.

3. Propose a method to temporally align the language and video modalities to allow for the spreading of language over time using recently tested methods for getting temporal information into and out of an LM (Perez, M., et al, 2022).

The intuition of this architecture is to condition the LM's output on the state of the environment (preprocessed by the agent) so that its estimations can be more specific and useful to the agent's situation, and to then in turn condition the agent's actions on the LM's representations to improve its decision making. While a separate visual encoder could be fed to the LM to condition it on the environment state, fetching video data from the agent's existing video processing section reduces parameter count, and it is hoped that its representations will have stronger priors for making decisions in the action space and speed up learning, as opposed to learning these domain specific representations by training a separate vision encoder. The LM could be placed directly after the CNN section, but instead, video tokens are allowed to be processed by the first transformer layer in the agent before being passed to cross attention and the LM; the further processing of CNN tokens in the first transformer layer of the agent may bring out more general abstractions from the video tokens that match better closer LMs general representations. The more similar architecture may also improve learning efficiency, but due to compute constraints no ablations are done into measuring the impact of drawing tokens from here as opposed to the CNN. The language model could be shifted further up in the agents transformer, but it is important, that the agent be able to consider the LMs outputs early enough in its layers that its actions can be meaningfully conditioned on them.

Fusion Method While many architectures for visual language models exist, few exist for video and language understanding. Within this niche, existing architectures are quite

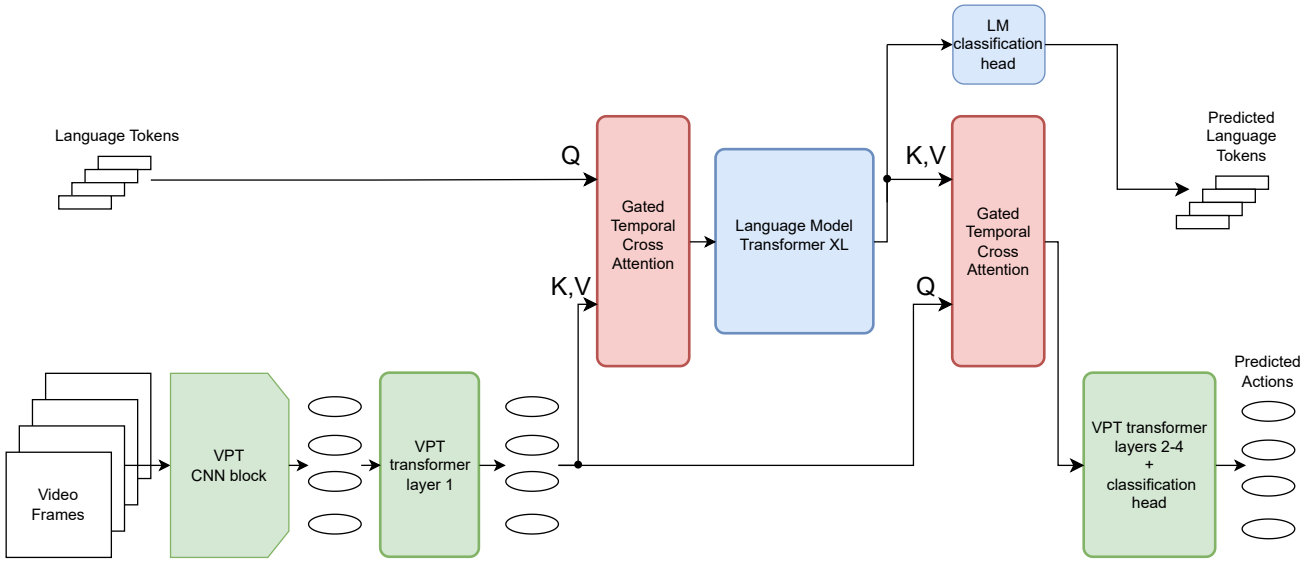

Figure 3.1: **Architecture Diagram -** an overview of the architecture proposed. In green is the original VPT agent model. In blue is the language model being fused in. In red is the cross attention mechanism permitting the fusion of the different modalities. Specifics like residual connections and most individual dense layers are not shown.

varied (Chen and Jiang, 2021), (Juncheng, Siliang and Linchao, 2021), but as with other VLMs, the current state of the art lies with transformer models, such as GIT (Jianfeng, Zhengyuan and Xiaowei, 2022), Flamingo, CoCa (Vasudevan, Legg and Seyedhosseini, 2022) and VIOLET (Fu, Linjie and Zhe, 2022). Most of these works take advantage of multiple uni-modal models and fuse them; since this allows for embodying a pretrained language model, and we need to take advantage of pretrained models here for compute reasons (since training from scratch is costly), we will be focusing on these. There are two main competing way to fuse models: cross attention and concatenation. Concatenation involves placing tokens from a new modality into the context of a model previously trained on a different modality. This is used in GIT. The GIT paper finds that this method is poor for taking advantage of pretrained models - when they concatenate visual tokens into a BERT model's language input, they find no performance difference between using newly initalised weights and pretrained weights. This suggests that concatenation does not lend to the model being able to use its pretrained weights, likely needing to catastrophically forget its previous weights and create completely new representations for the new modality.

The fusion method of this paper is therefore based on cross attention, and more closely on the PaLI and Flamingo VLM architectures (while CoCa uses cross attention, it uses a third model in addition to two pretrained unimodal ones, which is trained from scratch). The gated cross attention mechanism introduced in by Flamingo is very attractive for it's stability and ability to take advantage of pretrained models. Cross attention permits attention connections from queries of one modality to key/values of a new modality.

While this would usually also introduce a large distribution shift to the language model, gated cross attention introduces a learnable gating parameter - a scalar which scales the values outputted by the cross attention. Since it uses skip connections over the attention layer and the gate is initialized at 0, two models which are joined by a newly initalised gated cross attention block act completely independently - the output from one model to the other is scaled down to 0. However, since it is learnable, backpropagation allows for the gate to be increased and allow information from the new modality into the model as it benefits the loss function. This allows a stable introduction of the new modality. PaLI's specific method of only using cross attention at the input layers is used here for ease of implementation.

Silence Tokens A key difficulty with attempting to fuse the language models with an agent is the fact that spoken words/tokens are very sparse an irregularly spaced across time.

A naive solution to this is to simply take in language tokens as they come and only give output to the VPT model when a new language token comes in. There are a few key issues with this: A) during inference in the environment, we now have no way to know when to output a token, since we were previously outputting them as they occurred in the ground truth data, rather than based on what the language model was predicting B) We still want the language model to take input from the environment and to give useful representations to the VPT model regularly to assist it. If an episode is silent, however, the language model will never give output to the VPT model, and never take input from the VPT model.

We need some way for the language model to A) predict when language tokens should be outputted or when to stay 'silent' B) regularly take input from the VPT model and give output even during silence.

Recent work in a similar vein of getting language models to work across time (Perez, Jaiswal and Minxue, 2022) does so with spoken speech by adding a 'silence token' into the tokenizer and inserting them into the language input wherever recorded speakers are silent for a given period of time. They then train a BERT (Devlin, Chang and Lee, 2019) language model to do language modelling with these additional silence tokens and find that it is able to effectively use these silence tokens for sentiment analysis. This success inspires us to use this silence token method to pad out the language signal during pauses, that way there are tokens being processed which can attend to the video signal and give relevant language output even without speech. To build on this, rather than add a new token to the tokeniser, we try to identify an already existing token that can be effectively used as a silence token to speed up learning. We insert silence tokens randomly into an extract from wiki-text-103, trying a few candidate tokens in the tokeniser as the silence token (including the newline character, a coma, elipses, a hyphen, and space), and find that the token which least impacts performance when used as the silence token is a space. We therefore use this as the silence token.

However, this has a major disadvantage that silence tokens now take up the majority of the context, removing language from the effective context length of the language model, while also shifting the input largely out of distribution from what it was previously trained on, which will harm performance. The average measured tokens per second of language in the collected dataset is around 200. This means that, on average, silence tokens take up over 80% of the context length. To mitigate this, we propose a timeout

rate for the language model, only allowing it to take input and give output every N timesteps. While this causes the language and video tokens to be sampled at a different rate they are still synchronised, and cross attention permits this variation. While the later VPT layers will get input from older LM outputs through cross attention, they do still have the correct positional embeddings since they are conditioned on the earlier VPT layer which adds this, so this shouldn't pose a learning problem. One of the key pre-processing steps when implementing this is to ensure that, when we skip over input language tokens, we avoid skipping over spoken tokens instead of silence tokens. To do this, we move language tokens that we would have skipped to the next nearest timestep at which the LM is active - we ensure to only move tokens from previous timeteps to future timesteps to maintain causality. This approach unfortunately ends up causing videos with a amount of high words per minute (WPM) to pile up many tokens spoken in succession and spread them across a large range of time. This caused instability in training and so we stick with a timeout rate of 1. While we could remove higher WPM videos, our dataset is not large enough to permit this. However, we suggest exploring this timeout rate in future work in embodiment with larger datasets, partially because it makes intuitive sense as a way to do this type of transfer learning while reducing the distribution shift, and because the idea is reminiscent of the use of multiple timescales in multiscale recurrent neural networks, which have been shown to improve performance at with long term dependencies (Jaderberg et al., 2019)

Transformer-XL Language Model Because the agent runs for an arbitrary amount of time, the language model must be recurrent in order to not hit a limited context length and afterward have to be used in a strided manner across the input, which is prohibitively computationally expensive. The Transformer-XL (Dai et al., 2019) mechanism permits this via a relative positional encoding scheme and a recurrent cross-attention mechanism which enables each token to only require a single pass through the model regardless of input length, making training and inference much more efficient. This mechanism also allows the model to make use of longer term dependencies than the self-attention context size alone permits. We therefore use weights distributed by the original Transformer-XL paper on a GitHub repository (Yang et al, 2019) (specifically, the 300M parameter model trained on WikiText-103, as available on HuggingFace (HuggingFace, 2019)), since this was the largest pretrained transformer-XL language model found, and through prompting was found to already have knowledge of Minecraft, hopefully enabling some amount of transfer learning to speed up learning).

Gated Recurrent Cross Attention For the same reason as with the language model, the cross attention mechanism between the two models must also be recurrent. This is done simply by having each query attend to the most recent N keys and values, and removing keys/values beyond this context length. This avoids having each query attend to an arbitrary number of past elements, which becomes infeasible for long sequences. Below is the formula for this form of cross attention, given for a single query at time step t with a context of V keys/queries (since the keys and values are the same in cross attention, they are referred to as singular V here). For stability, and for measuring the dependence of models on the new inputted modalities, we use gated cross attention from the Flamingo paper, with learnable gates initialised at 0 so that, at initialisation, the

models behave entirely separately and do not receive out of distribution inputs, but can steadily learn to condition on each other as training progresses. As with transformer-XL training, keys and values from previous batches are accessible to the model during training and have gradients removed

$$\text{Temporal Cross Attention}(Qt, V_{0:t}) = \text{softmax}\left(\frac{Q_t V_{t-N:t}^T}{\sqrt{d_v}}\right) V_{t-N:t}$$

It is important that the language model give language output which is related to the agent's situation; this requires conditioning it on the environment state The tokens outputted from the transformer layer just after the video encoding section of the agent (CNN and Dense layers) are put through a gated cross attention block and fed to the language model (gated cross attention blocks are modified to include transformer-XL style recurrence). The LM's final transformer layer outputs are then put through a second recurrent gated cross attention block and fed back into the agent's second transformer layer, while the LM's standard classification head is used to predict the next word as usual. In this way, the language model is conditioned on the agent's early activations (corresponding to the state of the environment), and the agent is conditioned on the language model's output activations. To the best knowledge of the researcher, this is the first time that a mutually-conditioning LM-agent fusion method is described.

Since this temporal cross attention is a key contribution of the paper and it is not intuitively clear from the overall architecture diagram how it works, below is a diagram showing how it works for a specific set of tokens over time, including silence tokens and some preprocessing steps.

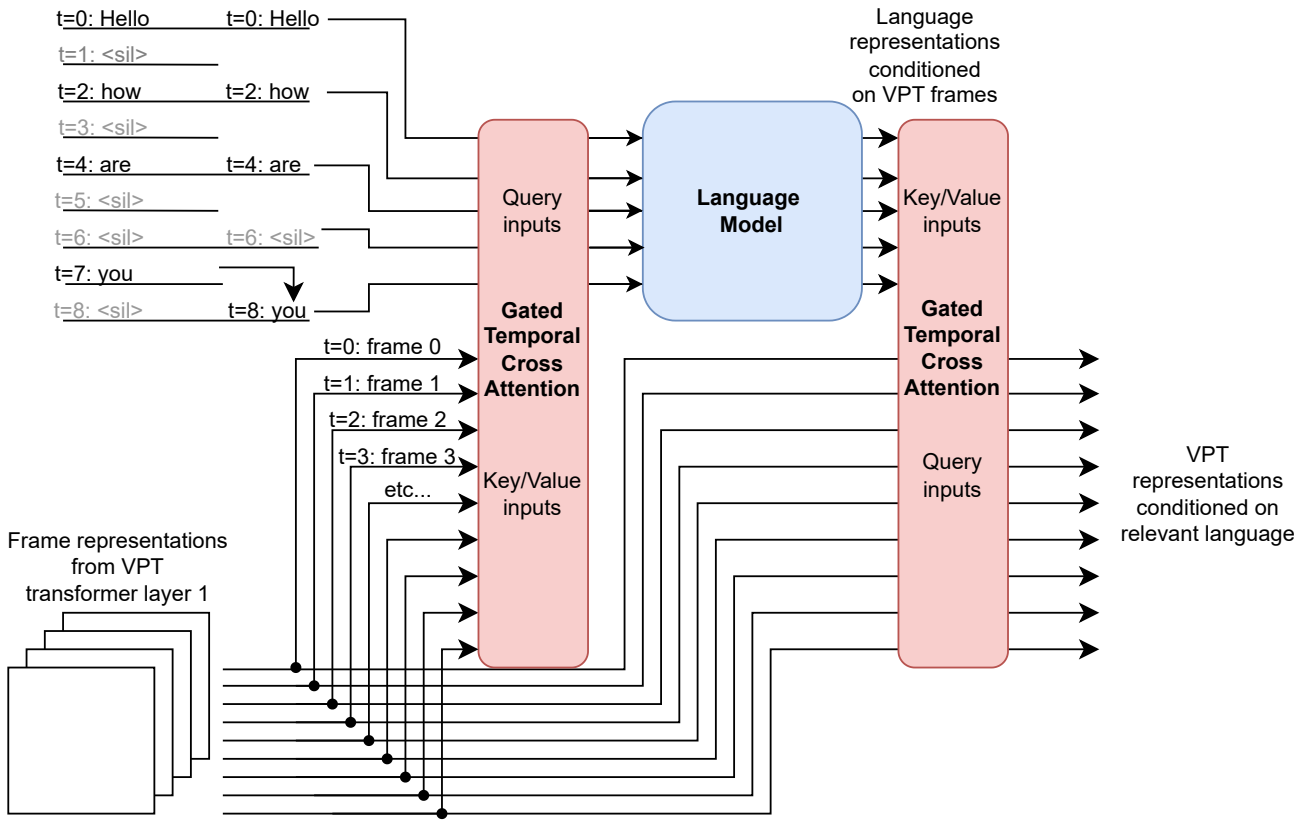

Figure 3.2: **Causal text/video alignment via temporal cross attention and silence tokens -** A detailed view of how cross attention is done across time, how silence tokens are used to align text and video and how the language model timeout rate is applied - a timeout rate of 2 is used above. In the actual code it is 4. To be clear, when 'frames' are mentioned in the diagram above, it refers to hidden representations taken from the output of the first transformer layer of the VPT agent. The exact causal masks used here are shown below

The two tables show how the causal mask is constructed for truncated/recurrent cross attention in the first and second cross attention blocks for the example inputs in the previous diagram, with the language model timeout rate as 2. Cells in red show where a mask is placed to maintain causality, cells in orange masked to allow for recurrence, in order to keep a consistent amount of keys/values in view to each query. It is shown coming into affect much earlier than usual here - these tables use a temporal cross attention memory length (Cn) of 4, while in code we use a length of 128, so the orange cells would be much further to the left. In blue are cells with indices which are permitted to cross attend. In the code, the first Cn queries each still see Cn past keys/values despite there being no past frames because they initialised to zero. As before, when 'frames' are

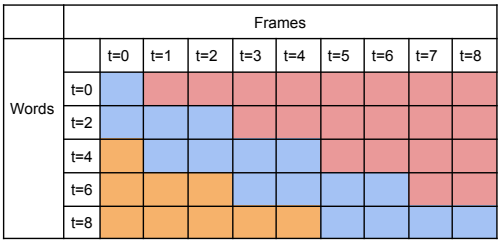

Figure 3.3: Attention mask for first cross attention block

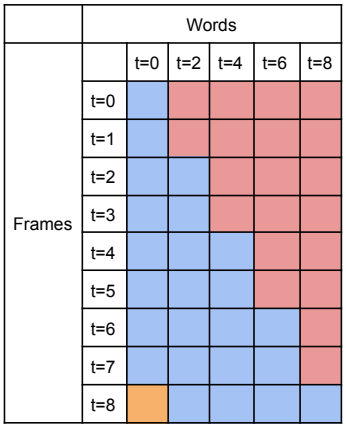

Figure 3.4: Attention mask for second cross attention block

mentioned above, we mean frame embeddings from VPT transformer layer 1.

Inference in Minecraft During inference in the MineRL environment, the agent will generate an action every timestep, and a language token every D timesteps, where D is the language model timeout rate. The estimated language token will be added back into the language model's context as per usual LM auto-regression. This is one of the key considerations of this architecture: permitting, after training, for simple transformer-XL style auto-regressive inference within the environment while fusing the two modalities.

Unfortunately, due to difficulties with reproducing the original VPT paper results using the provided VPT code and erratic behaviour caused by training with it, we are unable to test in the environment due to erratic behaviour. The code does, however, work correctly for training to predict action labels.

3.2 Data Collection

Data collection was initially done in the same manner as the original VPT paper, including acquiring a dataset of random frames which were sorted into clean single player survival Minecraft video, and other (modded Minecraft, multiplayer, creative mode, non-Minecraft video...) and training an SVM classifier on top of a CLIP to identify clean videos online. However, the classifier was not as performant as desired, and with the compute constraints for training the target dataset size would be small, so it was important that it was clean, and it was realised that the dataset size limits were small enough to allow for manual video labelling from the list of Minecraft videos provided by MineDojo (MineDojo Team, 2022) with some text filtering.

A small application with decent frame retrieval speeds and simple UI was developed to speed up labelling. We additionally had to ensure videos were in English for use with the English-only language model, and that the words-per-minute of spoken words was close to the target WPM of 180.

For automatic speech recognition, we use OpenAI's Whisper (OpenAI, 2022) with Stable-ts (Jianfch, 2022) to get precise word-level timestamps, and use a custom script to translate between the Whisper tokeniser and the language model's. After collecting this dataset, the VPT github program to run the inverse dynamics model (a highly accurate non-causal model for predicting actions from Minecraft video, provided by the VPT paper) was modified to run according to the specifics of the paper, save the actions this to disk, and improve efficiency by batching the input. In the end we collected a dataset of 172 valid videos of an appropriate WPM, transcripts and IDM action labels. In order to allow for long term language correlations to be made, we do not crop the videos, and we use a minimum video length of 7.5 minutes (this number is a hangover from an initial, different architecture which had a maximum context length of 2048 and would filter out silence tokens from the input, making its context window take up about 7.5 minutes)

In order to create a behavioural cloning dataset from the videos, we need to generate action labels for each frame accurately. How to do this is one of the key offerings of the VPT paper. This would regularly require gathering contractor data of Minecraft gameplay and training the IDM to predict actions non-causally from video, but are able to take advantage of the pretrained inverse dynamics model provided by the VPT repository. More information on modifications to the IDM labelling pipeline are in the Dataset and Code sections of the appendices.

3.3 Training

While we initially train the models together as a single joint model, we face two difficulties: 1. Training the joint models requires training on a singular loss, which is the sum of the language modelling loss, and the action modelling loss. This causes competition between the language and action cloning objectives and results in competing gradients which may cause training instabilities. 2. This joint model takes up a large amount of VRAM, resulting in having to use small sequence lengths and batch sizes, we might harm final accuracy.

We therefore split training into two phases: training the language model to condition on early VPT model layers, and training the later VPT layers to condition on the language model. This still results in a mutually conditioning architecture.

When training the language model, we freeze the VPT layers leading up to it (i.e. the CNN block and the first transformer layer) and train the LM and the first recurrent cross-attention block with the language modelling loss. Since video is still being passed through the frozen VPT layers leading up to it, the language model can learn to condition its output on the agent's representations. To train the later VPT layers, we then freeze that joint model and train the remaining VPT layers and the second cross-attention layer to predict the next action from the output of the first VPT layer as well as the language model's output. While the VPT training still requires having all parameters loaded (as the LM training phase uses the earlier VPT layers), since they are not being trained, we can run data through them without tracking gradients or loading their parameters into the optimizer, greatly lowering VRAM and computation usage.

This methodology also makes it easier to attribute improvement in performance to multimodality; if we were training the language model with a term in its loss function

for action prediction via the VPT layers, it may simply route frame data from the first VPT layer through itself to the later VPT layers, training the LM as if it were extra parameters for the VPT model, rather than having the final VPT layer learn to use its language abilities.

Both models are loaded in 16-bit bfloat to reduce memory use - the provided VPT repository code is modified to allow for this quantization. We find that this quantization minimally impacts performance.

For training the language model, we are able to use an active sequence length of 96 with a batch size of 4. Through the transformer-XL mechanism, the model is able to see the keys/values in the previous batch as well. We use a learning rate of 0.0001. As with the original transformer-XL code, we use Adam with no weight decay.

For training the VPT model, we are able to use an active sequence length of 128 and a batch size of 16, since this model is much smaller than the language model (120M parameters as opposed to 500M). We use a learning rate of 0.0001 and weight decay of 0.039.

Between the optimisations made mentioned above and the relative efficiency of the proposed architecture, we are train the model with only 15GB of VRAM.

Of important note is that during training we faced difficulties reproducing the VPT paper behaviour, which caused us to be unable to test in the MineRL environment; the original VPT paper reaches a loss of around 2-3 with the foundation models on the data scraped from the web, but with the data we scraped from the web, and testing with the contractor data provided in the VPT GitHub repository, we achieve a loss of around 8-12 using the provided behavioural cloning code. We suspect there is some extra detail in the training pipeline in the VPT paper that is not replicated with the provided code, but could not find it. The effect is that the effective loss function is shifted from the original VPT paper, causing the VPT trained models to become out of distribution. This would subsequently cause the agent to act erratically. To reduce this effect, we add a KL divergence loss term to between the original model. It additionally means that we will see training loss drop subsantially for the VPT model simply by training it at all, unrelated to the architecture. We will also see a drop in the loss of the language model as it shifts from predicting Wikipedia text to Minecraft-related language and silence tokens.

Dropout regularisation is not mentioned in the VPT paper and is not present in the code from the VPT repository, but as it is a standard in modern deep learning it is assumed to have been used implicitly, and so it is re-implemented into the code base after each transformer block, as the Transformer paper details. From the VPT repo, several changes are made to make it ready for larger training runs, including: permitting batch sizes greater than one; editing the data loading and training pipeline so that long sequences of actions (as opposed to a single timestep) can be loaded and trained on in a single batch, as per the proper transformer-XL training methodology; adding a cosine learning rate schedule with warmup (400 steps) and various other tweaks. This all builds on top of the existing example training code provided for by the VPT paper used in this paper is provided (save for the code used for mining the videos); preprocessing videos into chunks as Numpy files for faster training without GPU idling; permitting .mp4 frame rates other than 20 for easier data proccessing pipeline; cropping black borders from videos and resizing appropriately.

Of important note is that the provided VPT repository has differences from the original paper which prevent us from replicating their results. The result are a noisier model which performs erratically in the actual MineRL environment. However, the training pipeline does still optimize for predicting the next action. While unfortunately this prevents us from evaluating the agents behaviours in the environment, measuring the losses is still an indication of how the architecture improves performance, and the results for this can be applied to other models and training pipelines which do not have the issues faced here. Due to this issue, the training and validation loss are not useful, because decreasing loss does not suggest the language is improving performance, the decrease in loss is caused by the model being trained for this out-of distribution training pipeline. They are shown regardless below

Results

4.1 Language modelling

This is the first stage of training, and for this we fuse the with provided VPT agent and the provided Transformer-XL language model.

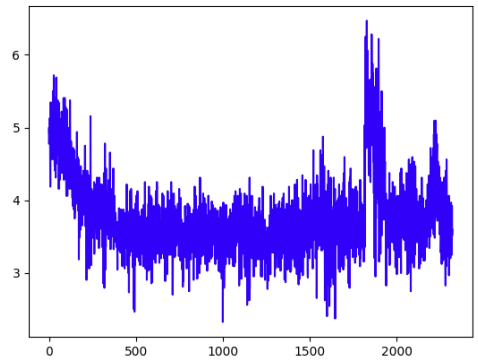

Figure 4.1: Training loss: language perplexity against batches

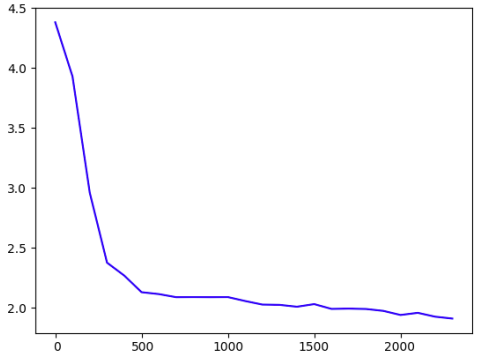

Figure 4.2: Validation loss: language perplexity against batches

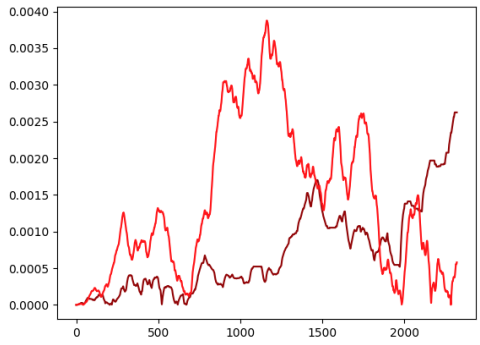

Figure 4.3: Cross Attention gate values for first cross attention block

Figure 4.4: Difference between language perplexity with obfuscated/correct video

Cross attention gate value Since all information passing between the models must go through the gate in the cross attention block, we can see how much information is crossing between the two models by looking at the absolute value this gate. Since it is a learnable parameter, if the signal from one model is useful to another, it can becomes larger to enable more signal to get through.

In Figure 4.3 the dark red curve plots the absolute value of the gate over the cross attention layer, while the light red curve corresponds to the absolute value of the gate over the dense MLP layer after cross attention.

Testing the Language Model's Video Dependence In order to ascertain that the LM is dependant on the video input, we need to remove the video signal and see if performance is affected. Doing this by masking it in any way is out of distribution for the model and may therefore impact performance unfairly. In order to obscure video information while providing in-distribution input, we measure performance on the evaluation set while swapping the video signals between different episodes. This way the model cannot depend on the video for estimating language since they are uncorrelated. Here is plotted the difference between the language modelling perplexity with and without videos swapped (higher values means the network performed better with the original video signal)

4.2 Behavioural cloning

This is the second stage of training. For this we fuse the with provided VPT agent with the Transformer-XL language model trained to be conditioned on video in the previous section.

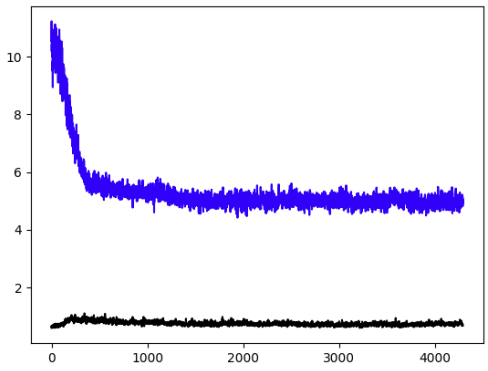

Figure 4.5: Training loss: action log loss against batches in blue. KL-divergence is shown in black.

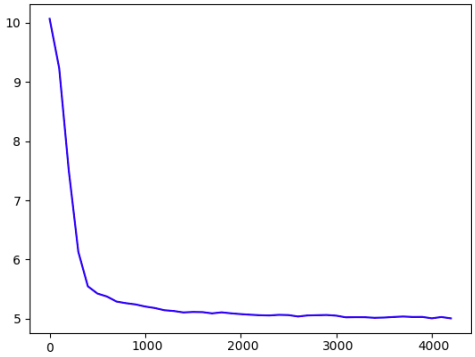

Figure 4.6: Validation loss: action log loss against batches

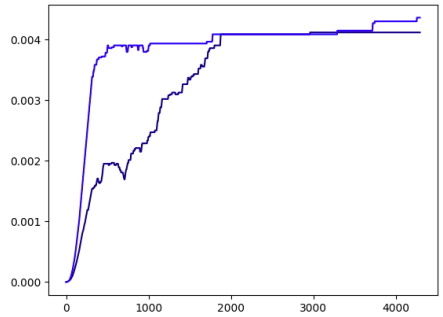

Figure 4.7: Cross Attention gate values for second cross attention block

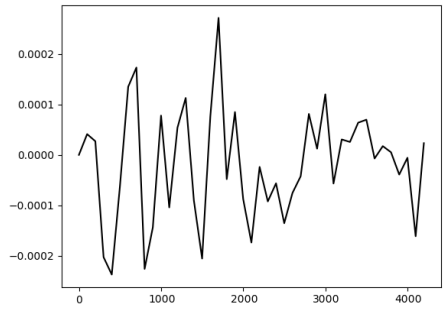

Figure 4.8: Difference between action log loss with obfuscated/correct transcripts.

Cross attention gate value In Figure 4.7 the dark red curve plots the absolute value of the gate over the cross attention layer, while the light red curve corresponds to the absolute value of the gate over the dense MLP layer after cross attention.

Testing VPT's Language Dependence We run the same dependence test as before, but swapping transcripts instead of video signals

As can be seen from the graphs above, we do indeed see the cross attention gates increase across training, especially for the VPT model's gates when conditioning on the language model. While in the Flamingo paper this indicated cross-modality dependence, we are not able to see any significant dependence in this model using our obfuscated modality metric. We also evaluate the loss for VPT action predictions with all words

masked with a silence token, and also find no statistically significant increase in accuracy. While there are possible interpretations of the graph that support increased dependence (both dependence graphs become more positive after an initially chaotic beginning, suggesting learning) they do not seem statistically justifiable. We suspect either a bug in our code or not having enough training data - the language experiment in the VPT paper uses 17,000 hours of video, a 200x difference, and the effect on the agent's ability to be conditioned on language is, in their own words, 'weak', needing to be measured for statistical significance over many trials. This difference in dependence is not directly comparable - it is likely much harder to train the agent to act in the environment than to measurably affect the loss on the behavioural cloning dataset, but it illustrates that this is a difficult task.

4.3 Discussion & Future Work

In light of the lack of meaningful statistical significance in these graphs, but considering the small amount of data and compute used (these models were training on only 80 hours of data a single T5 GPU over 5 hours), we are left with inconclusive results about the performance of this architecture. It would be very insightful to see this architecture and training methodology applied at a larger scale and with a larger team able to more thoroughly inspect the implementation for bugs.

Unfortunately, both due to this small scale and the difficulty in reproducing the training pipeline from the original VPT paper, testing for intelligent behaviour by the agent in the Minecraft environment was not possible due to erratic behaviour. An improved training pipeline is necessary to do evaluation of the agent in an actual environment rather than the training setting. This way the key questions around interpretability offerings and long term planning, as well as steerability could be answered.

For now, this architecture and these training losses are left for future research to investigate what the exact issue was here. The code will be publicly distributed so that it will not have to be re-implmemented, and to further incentivise work here.

R1. we are able to create an efficient architecture to do the kind of fusion we had in mind. it has the same time complexity as a regular transformer-XL network, able to produce a token in a single forward pass through each layer. While the model is large because of joining two large models (500M parameter LM and 240M parameter VPT), we are able to mitigate this using float16 conversion. The model is stable - in our tests, our joint model is able to reach similar loss performance to the unmodified VPT architecture, showing that it does not suffer from instability or other quirks due to its design. We are able to propose an architecture which minimises distribution shift so we can take efficiently advantage of both models pretraining - gated cross attention initialised at 0 ensure this is the case for modality mixing, but getting the language model to consistently take in data from the video modality across time required silence tokens and mitigating that using the LM timeout rate. That this is able to work well enough to do a serious training run without such instabilities is in our minds a strong contribution to the field of language model embodiment

R2. We are able to collect such a dataset and an efficient pipeline to collect 500 videos over a few days. We are succesfully able to use ASR software to create word-level timestamps and are able to translate word-level timestamps into token-level timestamps

for the language model to work with in sync with the VPT model such that causality is preserved. We are able to re-implement the IDM labelling method used in the VPT paper and improve its efficiency with batching, and identify that labelling with the IDM is somewhat costly in terms of GPU time and is a limiting factor in collecting data.

R3. While we show that the model is trainable and improves both language and action prediction performance, we are unfortunately unable to see that the language input causally affect the agents ability to predict actions, or that the language model learns to depend on the representations from the VPT model. It is unclear whether this is because of a lack of data; an issue with the architecture; or a bug in the training pipeline, and due to time constraints we leave this uninvestigated, as an inconclusive result.

R4. Being unable to test the model in the MineRL environment is unfortunate, and means we leave R3 and R4 unanswered. Scaling up this work in future research would

Much of the effort going into this research paper was around the engineering, creating a suitably clean and timely data collection pipeline, reverse engineering the VPT codebase so that i

4.4 Ethics

"Does your project involve people for the collection of data other than you and your supervisor(s)?" YES - we collect a web scrape of many hours of publicly available YouTube videos. We ensure to comply with legal requirements surrounding data scraping, copyrighted material and the terms of services of the relevant parties. This is discussed further in the separate ethics documentation file.

Copyright Due to the extensive development of VPT, the availability of their code and the very similar nature between this work and that work, rather than code from scratch this work creates a branch from this work, extending the codebase. No claims are made that the original codebase is our own work, and we clarify which parts we contributed to in the appendices.

Due to the need for large quantities of video from YouTube, specifically Minecraft gameplay videos, are downloaded and trained on. We ensure compliance with UK copyright and data mining law as well as YouTube's usage policy. This work credits the creators and platform owners; a file is kept in the provided Data Gathering folder of the dataset of the links to the videos used.

Due to the usage of the Minecraft environment and the MineRL environment built on top of it, both Minecraft and MineRL are recognised for their own work.

Ethical implication of this paper This paper makes use of natural language generation. The language model used may, either during its pre-training or during the training of the joint model on the gameplay captions, may have been trained on biased, toxic or otherwise harmful data, and as a result may generate text with these issues.

This paper concerns itself in the domain of advancing behavioural cloning to advance agent performance and speed of learning new environments. The ideas in this paper may be used to make artificial agents which are more competent at tasks which benefit society and are used for such purposes, but conversely may be used to produce agents used for

tasks which are harmful to society; the dual use problem. Especially because this allows agents to be instructed by language, it allows for very general use cases, and there may be a situation in which an agent is asked in language to do something harmful. Conversely, improvements in RLHF (OpenAI, 2023) make language models especially alignable to human values, and the potential for this model to be interpretable offers further AI safety benefits. We therefore believe that further work into language model embodiment is promising from an AI safety research perspective, and hope that this research leads towards architectures which lend themselves more easily to alignability.

Bibliography

[1] Alayrac, Jean-Baptiste, Jeff Donahue, and Pauline Luc. 2022. "Flamingo: a Visual Language Model for Few-Shot Learning." *arXiv*. https://arxiv.org/abs/2204.14198v1.

[2] Baker, Bowen, Ilge Akkaya, and Peter Zhokhov. 2022. "Video PreTraining (VPT): Learning to Act by Watching Unlabeled Online Videos." *arXiv*. https://arxiv.org/abs/2206.11795.

[3] Chen, Shaoxiang, and Yu-Gang Jiang. 2021. "Motion Guided Region Message Passing for Video Captioning." In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021. N.p.: The Computer Vision Foundation. https://openaccess.thecvf.com/content/ICCV2021/papers/Chen_Motion_Guided_Region_Message_Passing_for_Video_Captioning_ICCV_2021_paper.pdf.

[4] Chen, Xi, Xiao Wang, and Soravit Changpinyo. 2022 "PaLI: A Jointly-Scaled Multilingual Language-Image Model." *arXiv*. Accessed November, 2022. https://arxiv.org/abs/2209.06794v2.

[5] Devlin, Jacob, Ming-Wei Chang, and Kenton Lee. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *arXiv*.

[6] Kıcıman, E., Ness, R., Sharma, A., Tan, C. 2023. "Causal Reasoning and Large Language Models: Opening a New Frontier for Causality." *arXiv*. https://arxiv.org/abs/2305.00050.

[7] Gus, William Guss H., Brandon Houghton, and Nicholay Topin. 2019. "MineRL: A Large-Scale Dataset of Minecraft Demonstration." *arXiv*. https://arxiv.org/abs/1907.13440v1.

[8] Jianfeng, Wang, Yang Zhengyuan, and Hu Xiaowei. 2022. "GIT: A Generative Image-to-text Transformer for Vision and Language." *arXiv*. https://arxiv.org/pdf/2205.14100.pdf.

[9] Juncheng, Li, Tang Siliang, and Zhu Linchao. 2021. "Adaptive Hierarchical Graph Reasoning with Semantic Coherence for Video-and-Language Inference." *arXiv*. https://arxiv.org/abs/2107.12270v2.

[10] Michael Ahn, Anthony Brohan, and Noah Brown. 2022. "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances." *arXiv*. https://arxiv.org/abs/2204.01691v2.

[11] Perez, Matthew, Mimansa Jaiswal, and Niu Minxue. 2022. "Mind the gap: On the value of silence representations to lexical-based speech emotion recognition." In *Proceedings of Interspeech 2022*, Incheon, Korea: Interspeech. `https://www.isca-speech.org/archive/pdfs/interspeech_2022/perez22_interspeech.pdf`.

[12] Radford, Alec. 2021. "Learning Transferable Visual Models From Natural Language Supervision." *arXiv*.

[13] Radford, Alec, Jeffrey Wu, and Rewon Child. 2019. "Language Models are Unsupervised Multitask Learners." *OpenAI blog*. `https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

[14] Reid, Machel, Shixiang S. Gu, and Yutaro Yamada. 2022. "Can Wikipedia Help Offline Reinforcement Learning?" *arXiv*. `https://arxiv.org/abs/2201.12122`.

[15] Rosin, Guy D., Ido Guy, and Kira Radinsky. 2022. "Time Masking for Temporal Language Models." *arXiv*, https://arxiv.org/pdf/2110.06366.pdf.

[16] Huang, Shih-Cheng, Anuj Pareek, and Saeed Seyyedi. 2020. "Fusion of medical imaging and electronic health records using deep learning: a systematic review and implementation guidelines." N.p.: *npj Digital Medicine*.

[17] Stepputtis, Simon, Joseph Campbell, and Mariano Phielipp. 2020. "Language-Conditioned Imitation Learning for Robot Manipulation Task." https://arxiv.org/pdf/2010.12083.pdf.

[18] Sun, Chen, Austin Myers, and Carl Vondric. 2019. "VideoBERT: A Joint Model for Video and Language Representation Learning." *arXiv*, https://arxiv.org/abs/1904.01766v2.

[19] Brown, Tom B., Benjamin Mann, and Nick Ryder. 2020. "Language Models are Few-Shot Learners." https://arxiv.org/abs/2005.14165.

[20] Fu, Tsu-Jui, Li Linjie, and Gan Zhe. 2022. "VIOLET : End-to-End Video-Language Transformers with Masked Visual-token Modelling." *arXiv*, https://arxiv.org/pdf/2111.12681.pdf.

[21] Vaswani, Ashish, Noam Shazeer, and Niki Parmar. 2017. "Attention Is All You Need." *arXiv*, https://arxiv.org/abs/1706.03762v5.

[22] Wang, Zirui, Jiahui Yu, and Adams Wei Yu. 2021. "SimVLM: Simple Visual Language Model Pretraining with Weak Supervision." *arXiv*, https://arxiv.org/abs/2108.10904v3.

[23] Wei, Jason, Xuezhi Wang, and Dale Schuurmans. 2022. "Chain of Thought Prompting Elicits Reasoning in Large Language Models." *arXiv*, https://arxiv.org/abs/2201.11903v5.

[24] Wenhui, Wang, Bao Hangbo, and Dong Li. 2022. "Image as a Foreign Language: BEiT Pretraining for All Vision and Vision-Language Tasks." *arXiv*, https://arxiv.org/abs/2208.10442v2.

[25] Wolf, Thomas, Lysandre Debut, and Victor Sanh. 2020. "Transformers: State-of-the-Art Natural Language Processing." In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. N.p.: Association for Computational Linguistics.

[26] Paischer, F., Adler, T., Patvil, V., et al. 2022. "History Compression via Language Models in Reinforcement Learning." *Proceedings of the 39th International Conference on Machine Learning, Proceedings of Machine Learning Research*, PMLR 162:17156-17185.

[27] Vasudevan, V., Legg Y., Seyedhosseini, M. 2022. "CoCa: Contrastive Captioners are Image-Text Foundation Models." *arXiv*. https://arxiv.org/abs/2205.01917.

[28] Huang, Wenlong, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman and Brian Ichter. 2022. "Inner Monologue: Embodied Reasoning through Planning with Language Models" *arXiv preprint arXiv:2207.05608*. https://arxiv.org/abs/2207.05608.

[29] Wang, Zihao, Shaofei Cai, Anji Liu, Xiaojian Ma and Yitao Liang. 2023. "Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents" *arXiv preprint arXiv:2302.01560*. https://arxiv.org/abs/2302.01560.

[30] Lucidrains. 2022. "Flamingo-pytorch." *GitHub*, https://github.com/lucidrains/flamingo-pytorch/blob/main/flamingo_pytorch/flamingo_pytorch.py (Accessed: November, 2022)

[31] MineDojo Team. 2022. "MineDojo: Building Open-Ended Embodied Agents with Internet-Scale Knowledge." *NeurIPS*. https://minedojo.org/. (Accessed: March, 2023)

[32] Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., Salakhutdinov, R. 2019. "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context." *arXiv*. https://arxiv.org/abs/1901.02860.

[33] OpenAI. 2022. "Whisper: Robust Speech Recognition via Large-Scale Weak Supervision." *OpenAI*. https://openai.com/research/whisper.

[34] Jianfch. 2022. "Stable-ts: Modifies OpenAI's Whisper to produce more reliable timestamps." *GitHub*. https://github.com/jianfch/stable-ts. (Accessed: January, 2023)

[35] Driess, D., Xia, F., Sajjadi, M. S. M., Lynch, C., Chowdhery, A., Ichter, B., Wahid, A., Tompson, J., Vuong, Q., Yu, T., Huang, W., Chebotar, Y., Sermanet, P., Duckworth, D., Levine, S., Vanhoucke, V., Hausman, K., Toussaint, M., Greff, K., Zeng, A., Mordatch, I., Florence, P. 2023. "PaLM-E: An Embodied Multimodal Language Model." *arXiv*. https://arxiv.org/abs/2303.03378.

[36] Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, [. . .], and Thore Graepel. 2019. "Human-level performance in 3D multiplayer games with population-based reinforcement learning.'' *Science* 364(6443):859-865. https://www.science.org/doi/10.1126/science.aau6249

[37] OpenAI. 2023. "GPT-4 Technical Report." *arXiv*. https://arxiv.org/abs/2303.08774.

[38] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 248-255). IEEE. https://www.image-net.org/.

[39] OpenAI. (2022). Video Pre-Training. *GitHub*. Retrieved from https://github.com/openai/Video-Pre-Training. (Accessed: November, 2022)

[40] Dai, Zihang, Yang, Zhilin, Yang, Yiming, Carbonell, Jaime, Le, Quoc V., Salakhutdinov, Ruslan. 2019. "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context." *GitHub*, https://github.com/kimiyoung/transformer-xl (Accessed: November 4, 2022).

[41] Hugging Face. 2019. "transfo-xl-wt103." *Hugging Face*, https://huggingface.co/transfo-xl-wt103 (Accessed: November, 2022).

Appendices

5.1 Dataset

5.1.1 Dataset statistics

Dataset overview	
Number of clean videos collected	500
Number of hours of video collected	250
Number of videos with clean transcripts	340
Number of videos with action labels generated	445
Final number of videos used for training	172

Access The dataset can be accessed through the following link: `https://drive.google.com/drive/folders/1YWhr73V3LowPszRV1nGUo8GknIikM-zC?usp=sharing` The parent folder should be shared immediately but it may take a minute before the 'DATASET' folder is visible.

_DISSERTATION.ipynb runs you through the workings of all the steps to carry out the code produced by the dissertation. This includes pre-requisites installation, video selection/collection, transcription, IDM labelling, pre-processing, training the different sub-models and evaluation.

In order for the main program, , to work, it must be opened through colab and given a way to access the _DISSERTATION folder (this will require a signed in Google account). To do this, it is first necessary to create a link from the shared _DISSERTATION folder to the top directory 'My Drive' in your main Google Drive account. 1. Right click on the shared folder '_DISSERTATION' (this can be found in your 'Shared With Me' folder in Google Drive) 2. Select 'Add a shortcut to Drive' 3. In the list that pops up, the top entry should be 'My Drive'. To the far left of this entry should be the 'Add' button. Press this 4. The menu should disappear and the link should now be added. When you run _DISSERTATION.ipynb (right click ¿ Open with ¿ Google Collaboratory), it should now be able to mount and access all the required files

5.1.2 Manual Clean Video Classifier UI

5.2 Training Details

We cast the weights to blfoat16 for both models to improve training efficiency and

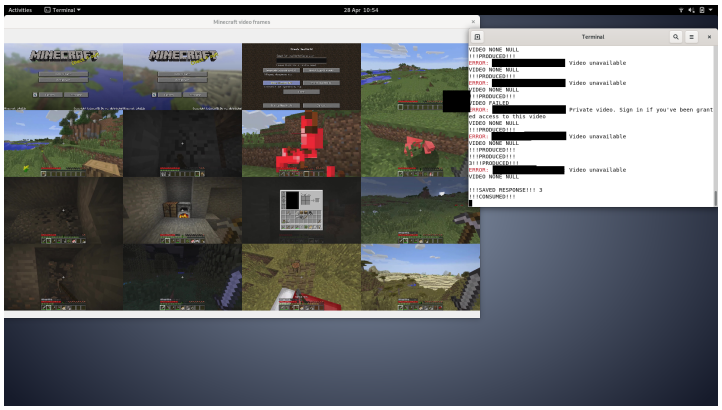

Figure 5.1: A screenshot of the manual video labelling UI produced for identifying clean Minecraft videos. Frames evenly spaced across the video are shown to teh user. A terminal shows the progress of video searching (e.g. if a video passing the meta data filters was found), and allows the user to input their response to if the video is clean (1), has artifacts (2) or is unclean (3).

allow us to use larger batch sizes

Proper shuffling of the dataset was crucial to stable training. For the fir

the provided VPT code had to be modified to remove batch accumulation and allow proper processing batch sizes and sequence lengths greater than one. It was modified to allow for proper tracking of if episodes start/end within each batch and masking past keys/values accordingly

In the original VPT paper, all null actions and frames that occur while performing them were removed from the behavioural cloning dataset to avoid mode collapse. This approach would not work in this paper, because the player may be speaking while performing no actions, and (in order to keep the frame and word streams temporally aligned) removing the null action would require us to remove the words as well, which corrupts the language input. As a compromise, we instead remove only null actions that also occur while no words are being said

5.3 Main Contributed Code

5.3.1 The key architecture implementation. Initialisation and Forward Method

```
1 def __init__(
2         self,
3         device,
```

```
        L_SEQ_LEN=256,
        LM_TIMEOUT_RATE=2,
        LM_type="transfo-xl-wt103",
        dtype=th.float32,               # allows for setting to bfloat16
        LM_ONLY=False,

        recurrence_type="transformer", # from VPT repo
        impala_width=8, # from VPT repo
        impala_chans=(16, 32, 32), # from VPT repo
        obs_processing_width=256, # from VPT repo
        hidsize=2048, # from VPT repo
        single_output=False,  # from VPT repo
        img_shape=[128,128,3], # from VPT repo
        scale_input_img=True, # from VPT repo
        only_img_input=False, # from VPT repo
        init_norm_kwargs={}, # from VPT repo
        impala_kwargs={'post_pool_groups': 1}, # from VPT repo
        # Unused argument assumed by forc.
        input_shape=None,  # pylint: disable=unused-argument # from VPT
            repo
        active_reward_monitors=None, # from VPT repo
        img_statistics=None, # from VPT repo
        first_conv_norm=False, # from VPT repo
        diff_mlp_embedding=False, # from VPT repo
        attention_mask_style="clipped_causal", # from VPT repo
        attention_heads=16, # from VPT repo
        attention_memory_size=2048, # from VPT repo
        use_pointwise_layer=True, # from VPT repo
        pointwise_ratio=4, # from VPT repo
        pointwise_use_activation=False, # from VPT repo
        n_recurrence_layers=4, # from VPT repo
        recurrence_is_residual=True, # from VPT repo
        timesteps=128, # from VPT repo
        use_pre_lstm_ln=False, # from VPT repo
        **unused_kwargs,
    ):
        super().__init__()
        self.device = device
        self.dtype=dtype



        print("Loading VLPT CNN and VPT with dtype:",dtype)

        self.L_SEQ_LEN = L_SEQ_LEN
        self.F_SEQ_LEN = L_SEQ_LEN*LM_TIMEOUT_RATE
        self.LM_TIMEOUT_RATE=LM_TIMEOUT_RATE# referred to also as 'D'. LM
            gets input from VPT and gives output to VPT every D timesteps
            to reduce silence token noise.
        self.LM_type = LM_type
        self.LM_ONLY=LM_ONLY
        self.XATNN_MEMLEN=512

        if LM_ONLY==True:
          print("W: LOADING IN LM_ONLY MODE! only the following layers will
               be loaded: [VPT_CNN, VPT_transfo_1, VPT_LM_Xattn, LM.")


```

```
        assert recurrence_type == "transformer"
        self.recurrence_type = "transformer"
        active_reward_monitors = active_reward_monitors or {}
        self.single_output = single_output
        chans = tuple(int(impala_width * c) for c in impala_chans)
        self.hidsize = hidsize

        # Dense init kwargs replaces batchnorm/groupnorm with layernorm # print
        self.init_norm_kwargs = init_norm_kwargs
        self.dense_init_norm_kwargs = deepcopy(init_norm_kwargs)
        if self.dense_init_norm_kwargs.get("group_norm_groups", None) is not None:
            self.dense_init_norm_kwargs.pop("group_norm_groups", None)
            self.dense_init_norm_kwargs["layer_norm"] = True
        if self.dense_init_norm_kwargs.get("batch_norm", False):
            self.dense_init_norm_kwargs.pop("batch_norm", False)
            self.dense_init_norm_kwargs["layer_norm"] = True

        # Setup inputs
        self.img_preprocess = ImgPreprocessing(img_statistics=img_statistics, scale_img=scale_input_img, dtype=dtype)
        self.img_process = ImgObsProcess(
            cnn_outsize=256,
            output_size=hidsize,
            inshape=img_shape,
            chans=chans,
            nblock=2,
            dense_init_norm_kwargs=self.dense_init_norm_kwargs,
            init_norm_kwargs=init_norm_kwargs,
            first_conv_norm=first_conv_norm,
            dtype=dtype,
            **impala_kwargs)









        # Define Language Model
        if LM_type is not None:
            self.LM = TransfoXLLMHeadModel.from_pretrained(LM_type).to(th.bfloat16) #, @later, load from stored weights
            self.LM.tie_weights() # necessary for this model)
            # @ CHANGE THESE DURING INFERENCE
            self.LM.reset_memory_length(self.L_SEQ_LEN)
            self.LM.transformer.same_length = True # eval:True
            self.LM.transformer.clamp_len = -1 # eval: 2000 - definitely experiment.



            # define cross atention layers from VPT->LM, LM->VPT
            self.Xattn_VPT_LM = MaskedGatedCrossAttention(
                embed_dim=self.LM.transformer.d_embed, # queries are lanuage tokens - LM gets appropriate size and number of tokens
                kvdim=hidsize, # embedding dimensions for VPT transformer layers
                num_heads=attention_heads,
```

```
                ffw_dim=hidsize*pointwise_ratio, # FFW hidden size should
                    be same architecturally as FFW in LM (i.e. ratio
                    between transformer token size and FFW hidden layer
                    should be same) (x2 to account for multimodal tokens)
                batch_first=True,
                dtype=self.dtype)

            if not LM_ONLY:
              self.Xattn_LM_VPT = MaskedGatedCrossAttention(
                  embed_dim=hidsize, # queries are VPT tokens - VPT gets
                      appropriate size and number of tokens
                  kvdim=self.LM.transformer.d_embed,
                  num_heads=attention_heads,
                  ffw_dim=hidsize*pointwise_ratio, # use same as FFW (x2 to
                       account for multimodal tokens)
                  batch_first=True,
                  dtype=self.dtype)




        if self.LM_ONLY:
          n_recurrence_layers=1
        # VPT transformer layers
        self.recurrent_layer = ResidualRecurrentBlocks(
            hidsize=hidsize,
            timesteps=timesteps,
            recurrence_type=recurrence_type,
            is_residual=recurrence_is_residual,
            use_pointwise_layer=use_pointwise_layer,
            pointwise_ratio=pointwise_ratio,
            pointwise_use_activation=pointwise_use_activation,
            attention_mask_style=attention_mask_style,
            attention_heads=attention_heads,
            attention_memory_size=attention_memory_size,
            n_block=n_recurrence_layers,
            dtype=dtype)

        if not self.LM_ONLY:
          self.lastlayer = FanInInitReLULayer(hidsize, hidsize, layer_type=
              "linear", dtype=dtype, **self.dense_init_norm_kwargs)
          self.final_ln = th.nn.LayerNorm(hidsize, dtype=dtype)


        self.dropout=th.nn.Dropout(p=0.2)
        self.w_embed_dropout=th.nn.Dropout(p=0.2)
        self.VPT0_VPT1_dropout=th.nn.Dropout(p=0.2)




def forward(self,
            ob_words=None, # inputted language tokens for langauge model.
                Silence tokens pad this to the temporal length of frame
                input
            ob_frames=None, # the inputted video frames to go to teh VPT
                model
            VPT_state=None,  # recurrent transformer-XL keys/values which
```

```
                preserve the hidden state across batches (for VPT)
            context=None, # for inference. hangover from previous
                architecture decisions. deprecated
            LM_state_in=None, # recurrent transformer-XL keys/values which
                preserve the hidden state across batches (for LM)
            LM_active_timestep=True, # for inference, since LM activity
                cannot be masked as it is in batches.
            LM_labels=None, # for training the LM.
            previous_LM_output=None, # for inference. deprecated.
            inference=False, # for inference. deprecated.
            stored_frames=None, # for inference. deprecated
            LM_ONLY=False, # Permits training just the langauge model alone
                , the following VPT layers are skipped
            Xattn2_state=None, # recurrent transformer-XL keys/values which
                 preserve the hidden state across batches (for 1st x-attn)
            Xattn1_state=None): # recurrent transformer-XL keys/values
                which preserve the hidden state across batches (for 2nd x-
                attn)
        BATCH_SIZE = ob_frames['img'].shape[0]



        # used by VPT to mask out past memories which the LM should not
            have access to in this batch
        # i.e. to mask out memories for index 0 if the sample at index 0 is
             the first few frames from a new video,
        # so that it cannot attend to frames from a previous video.
        first = context["first"]

        # the following if statement constructs the attention masks for the
             cross attention blocks if a language model is provided.
        # If when the model is initialise, LM_type is set to None, the
            model will operate as a plain VPT model without an LM.
        if self.LM_type != None:
          L_SEQ_LEN = ob_words.shape[1]//self.LM_TIMEOUT_RATE
          F_SEQ_LEN = ob_frames['img'].shape[1] # frame sequence length
          if inference:
            L_SEQ_LEN = 1 if LM_active_timestep else 0

          assert F_SEQ_LEN%self.LM_TIMEOUT_RATE==0 or inference

          Xattn1_mems_len=min(F_SEQ_LEN+Xattn1_state.shape[1], self.
              XATNN_MEMLEN+F_SEQ_LEN) # we do so that each token can see
              previous XATNN_MEM_LEN worth - we need to store XATNN_MEM_LEN
               previous from latest token and first token, which
          Xattn2_mems_len=min(L_SEQ_LEN+Xattn2_state.shape[1], self.
              XATNN_MEMLEN+L_SEQ_LEN)

          if inference:
            assert BATCH_SIZE==1 and F_SEQ_LEN==1 # doing RL inference
                requires batch size 1, timesteps 1
            if not LM_active_timestep:
                #self.Xattn_mask_in = th.ones([1,self.XATNN_MEMLEN+1]).to(
                    self.device) # this actually isnt used since the LM is
                    inactive
                self.Xattn_mask_out = th.ones([1,self.XATNN_MEMLEN]).to(
                    self.device) # during this timestep, VPT only takes
                    input from previous LM outputs, which are in its keys/
                    values cache (Xattn2_mems/VPT_state)
```

```
                #Xattn1_mems_len=min(F_SEQ_LEN+Xattn1_state.shape[1], self.
                    XATNN_MEMLEN+F_SEQ_LEN) # we do so that each token can
                    see previous XATNN_MEM_LEN worth - we need to store
                    XATNN_MEM_LEN previous from latest token and first
                    token, which
                Xattn2_mems_len=min(L_SEQ_LEN+Xattn2_state.shape[1], self.
                    XATNN_MEMLEN)
            else:
                self.Xattn_mask_in = th.zeros([1,self.XATNN_MEMLEN+1]).to(
                    self.device) # this actually isnt used since the LM is
                    inactive
                self.Xattn_mask_out = th.zeros([1,self.XATNN_MEMLEN+1]).to(
                    self.device) # during this timestep, VPT only takes
                    input from previous LM outputs, which are in its keys/
                    values cache (Xattn2_mems/VPT_state)
                Xattn1_mems_len=min(F_SEQ_LEN+Xattn1_state.shape[1], self.
                    XATNN_MEMLEN+1) # we do so that each token can see
                    previous XATNN_MEM_LEN worth - we need to store
                    XATNN_MEM_LEN previous from latest token and first
                    token, which
                Xattn2_mems_len=min(L_SEQ_LEN+Xattn2_state.shape[1], self.
                    XATNN_MEMLEN+1)

        else:
            assert (ob_words.shape[1])%self.LM_TIMEOUT_RATE==0, "for first
                timestep, word needs to see current frame. This offsets
                L_SEQ_LEN with F_SEQ_LEN: we have "
            # cross atention between every D langaueg tokens and every
                frame print20
            self.Xattn_mask_in=th.ones([L_SEQ_LEN, Xattn1_mems_len],
                requires_grad=False, dtype=th.bool).to(self.device) #
                create causal mask between language tokens and frames.
                every frame is paired with the previous word emitted, so
                that at the current frame we predic the next word from the
                current frame (and previous frames witha ttentions) and
                rpevious words. For more details look at agent.py
                words_to_agent() # since all sequeces in the batch (and
                every batch assuming constant D) have the same relation in
                terms of time with each other, all sequences can use the
                same Xattn mask
            for q in range(L_SEQ_LEN):
                end = -(q*self.LM_TIMEOUT_RATE+1)
                start = max(0, end-self.XATNN_MEMLEN)
                self.Xattn_mask_in[-q, start:end] = 0   # @ DOUBLE CHECK
                    THIS IS MASKING THE RIGHT WAY #@r2 - !if using
                    each_LM_token_attends_to_all_past_frames-style x-attn (
                    as opposed to only frames that occured during langauge
                    token), modify this mask appropriately. do +1 so LM
                    outputs word on first step so that it can be replicated
                     for the TIMEOUT steps
            # cross attention between every langauge output (including
                repeated tokens from LM durin LM timeout) and every frame
            self.Xattn_mask_out=th.ones([F_SEQ_LEN, Xattn2_mems_len],
                requires_grad=False, dtype=th.bool).to(self.device) #
                create causal mask between language tokens and frames. In
                the input, every frame is paired with the previous word
                emitted, so that at the current frame we predict the next
                word from the current frame (and previous frames witha
                ttentions) and previous words. For more details look at
```

```
                agent.py words_to_agent() # since all sequeces in the batch
                 (and every batch assuming constant D) have the same
                relation in terms of time with each other, all sequences
                can use the same Xattn mask
215             for q in range(F_SEQ_LEN):
216                 num_words_visible = int(q//self.LM_TIMEOUT_RATE)+1
217                 hidden_words = L_SEQ_LEN-num_words_visible
218                 start = max( -Xattn2_mems_len, -(1+hidden_words+self.
                        XATNN_MEMLEN) )
219                 end = Xattn2_mems_len-hidden_words
220                 self.Xattn_mask_out[q, start:end] = 0   # @ DOUBLE CHECK
                        THIS IS MASKING THE RIGHT WAY #@r2 - !if using
                        each_LM_token_attends_to_all_past_frames-style x-attn (
                        as opposed to only frames that occured during langauge
                        token), modify this mask appropriately.
221
222
223         with th.no_grad():
224             ### pass input frames through CNN section
225             # frames inputted as float32
226             x = self.img_preprocess(ob_frames["img"])
227             assert not th.isnan(x).any()
228             x = self.img_process(x).to(self.dtype)
229             assert not th.isnan(x).any()
230         #x = self.dropout(x) # VPT has dropout on all sublayer outputs, so
                internally it is dropoutted but needs input manually dropoutted
                 # --- this is never actually used. dopout is used ebfore a
                layer that needs dropout regularisation, but VPT transfo layer
                1 is always frozen. frozen during  LM training and VPT trainnig
                 to avoid disruptons what LM has fit to.
231
232         ### pass processed frames through VPT Agent transformer layer #1.
                make sure only 128 tokens are rpedicted with self attention at
                a time, otherwise trained differently to original (?) and BIG
                MEM.
233         x, [VPT_state[0]] = self.recurrent_layer(x, first, [VPT_state[0]],
                start_block=0, end_block=1)
234
235
236
237         # ---------------------------------------- INSERT LM
238         # during inference, we need to store frames made during LM_TIMEOUT
                so it can cross attend to these unseen frames when it is back
                on
239         if self.LM_type!=None: # we may want to use an original vanilla VPT
                 model to get its outputs for KL divergence comparison.
240             if LM_active_timestep:
241                 ### copy VPT_transofrmer_layer1 output to pass the LM via
                        cross-attention, leaving original output as residual
                        around LM (as in Flamingo-style gated cross-attention.
                        see: masked_gated_cross_atention.py)
242                 # implement LM timeout - D
243                 ### convert input word token ids to embeddings. since LM(
                        words) works from token_indices and we need to pass
244                 # VPT representations as raw vectors, we need to use LM(
                        words, from_embeddings=True),
245                 # which requires us to pre-embed to token ids
246                 x2 = x.clone()
247                 l = ob_words[:,::self.LM_TIMEOUT_RATE] # this applies the
```

```
                    LM timeout, only feteching every D language tokens. The
                     input should have been modifiedby the dataloader so
                    that we are only removing silence tokens.
248                 l = self.LM.transformer.word_emb(l).to(self.Xattn_VPT_LM.
                    dtype)
249
250
251                 ### Gated cross attention: FUSE output from VPT transformer
                     layer 1 and language
252                 l = self.w_embed_dropout(l) # apply dropout to language
                    embeddings
253                 Xattn1_state_out = th.cat([Xattn1_state.to(self.
                    Xattn_VPT_LM.dtype), x2.to(self.Xattn_VPT_LM.dtype)],
                    dim=1)[:,-Xattn1_mems_len:]   # concatenate previous
                    Xattn mems with current Xattn inputs. Use most recent [
                    XANN_MEMLN] of tokens as input
254                 fused_in = self.Xattn_VPT_LM(x=Xattn1_state_out.clone(), y=
                    l, attn_mask=self.Xattn_mask_in)
255                 assert not th.isnan(fused_in).any()
256
257                 #### Feed LM fused input tokens & predict raw LM output
258                 #### From raw LM output, predict word classes (for language
                     modelling loss)
259
260                 ### construct labels if possible
261                 if LM_labels!=None:
262                     #print("LM LABELS TOKENS", LM_labels)
263                     LM_labels = LM_labels[:,::self.LM_TIMEOUT_RATE] # since
                          input words are dropped according to D, labels are
                          dropped according to D, too
264                     #print("LM LABELS TOKENS STRIPPED", LM_labels)
265
266                 ### get LM raw output, word predictions and loss
267                 # transfo_xl uses dropout on the raw input embeddings.
                    since we are have already applied droppout via the
                    output of gated x-att, we removed it in the
                    modelling_transfo_xl.py file
268                 LM_words_out, LM_loss, LM_raw_out, LM_state_out = self.LM(
                    inputs_embeds=fused_in.to(self.LM.dtype), mems=
                    LM_state_in, labels=LM_labels, return_dict='VLPT',
                    output_hidden_states=True) # causal attention mask is
                    constructed internally. No need for padding despite
                    batch_size>1 because all sequences are always full (
                    always same number as input frames)
269
270                 # reshape LM_words
271                 LM_words_out = LM_words_out.view(BATCH_SIZE, l.shape[1],
                    -1).to(self.dtype)
272
273                 assert not th.isnan(LM_words_out).any()
274
275                 if self.LM_ONLY:
276                     return (None, None), LM_words_out, VPT_state,
                            LM_state_out, LM_loss, None, Xattn1_state_out, None
277
278             # the following code is deprecated. it is used for inference in
                     the MineRL environment, but due to issues with the VPT
                    repository and reproduceability, the model can only be
                    trained to output erratic behaviour. The code for inference
```

```
                is therefore left broken, since it wouldn't work anyway (
                it is left as it was before the temporal aspect to temporal
                 x-attn was added).
279             else: # During inferemce when forward() is called per timestep,
                     if LM is TIMEOUTed at this tiemstep, output previous
                    output from LM that VPT needs (does not include past words)
280                 LM_words_out = None # output during TIMEOUT is treated the
                        same as with silent tokens during TIMEOUT, and
                        discarded from LM input. dont need to predict it at all
                        .
281                 print('None')
282                 #top layer|all abtches|last token|full embeddings
283                 LM_loss = None
284                 LM_raw_out = th.zeros([1,0,self.LM.transformer.d_embed]).to
                        (self.device) # this
285                 LM_state_out = LM_state_in
286                 Xattn1_state_out = Xattn1_state
287                 #print('skipped LM cause inactive timestep') dropout
288
289
290             ### Gated cross attention: FUSE LM raw output & VPT-transformer
                    -layer-1 output
291             # fusing back with VPT-transformer-layer-1 output before going
                    into VPT layer 2 (and using gated cross-attention) means
                    that at the start of training, despit the added LM, the VPT
                     model is identical to the unmodified version, and
                    interaction between the two models can smoothly increase as
                     it is learnt to be useful (as in Flamingo). allows stable
                    training and avoid catastrophic forgetting.
292             # so language is passed as queries and VPT tokens are passed at
                     keys/values.
293             x = self.VPT0_VPT1_dropout(x)
294             Xattn2_state_out = th.cat([Xattn2_state.to(self.Xattn_LM_VPT.
                    dtype),LM_raw_out.to(self.Xattn_LM_VPT.dtype)], dim=1)[:,-
                    Xattn2_mems_len:]
295             fused_out = self.Xattn_LM_VPT(x=Xattn2_state_out.clone(), y=x.
                    to(self.Xattn_LM_VPT.dtype), attn_mask=self.Xattn_mask_out)
                    #now that word/frame queries/keys have beens waped, we need
                     to make a different attention mask to the first one
296             assert not th.isnan(fused_out).any()
297
298
299         else:
300             LM_loss=None
301             LM_state_out=LM_state_in
302             LM_words_out=None
303             fused_out=x
304             Xattn1_state_out, Xattn2_state_out = th.zeros([BATCH_SIZE,0,
                    self.hidsize]).to(self.device), th.zeros([BATCH_SIZE
                    ,0,1024]).to(self.device)  # if you want to run without a
                    language mode, change the ahrd coded model size here and in
                     BC_track_hidden.py
305
306         # ---------------------------------------- END INSERT LM
307
308
309         # pass combined tokens through remaining VPT transformer layers
                (2,3,4)
310         fused_out, VPT_state[1:4] = self.recurrent_layer(fused_out.to(self.
```

```
            dtype), first, VPT_state[1:4], start_block=1, end_block=4)

        # VPT PREDICT ACTION (for behaviour modelling loss)
        fused_out = self.lastlayer(fused_out)
        fused_out = self.final_ln(fused_out).to(th.float32)
        assert not fused_out.isnan().any()

        # format action and output
        pi_latent = vf_latent = fused_out
        #print('actions type',pi_latent.type(),pi_latent.type())

        return (pi_latent, vf_latent), LM_words_out, VPT_state,
            LM_state_out, LM_loss, previous_LM_output, Xattn1_state_out,
            Xattn2_state_out
```

5.3.2 truncated/recurrent cross attention

This is based off of the Flamingo paper's gated cross attention mechanism, and takes after the specific implementation by lucidrains, 2022. This code is located in lib/masked_gated_cross_attention.py

```
class MaskedGatedCrossAttention(th.nn.Module):

    def __init__(
        self,
        embed_dim,
        kvdim,            # for the cross atention block doing 'language +
            frame ->langauge model', img is passed through kv
        num_heads,    # heads in VPT-240M = 16
        ffw_dim,
        batch_first=True,
        device=None,
        dtype=th.float32
        ):

        super().__init__()

        self.attention = th.nn.MultiheadAttention(
            embed_dim=embed_dim,  # in pytorch, its split across heads
            num_heads=num_heads,
            dropout=0.0,
            bias=True,
            add_bias_kv=False,
            add_zero_attn=False,
            kdim=kvdim,vdim=kvdim,
            batch_first=batch_first,
            device=device,
            dtype=dtype)

        self.dtype=dtype
        self.ffw = MLP(insize=embed_dim, nhidlayer=1, outsize=embed_dim,
            hidsize=ffw_dim, hidactiv=sqrelu, dtype=dtype)
        self.num_heads = num_heads
        self.alpha_xattn = th.nn.Parameter(th.tensor(0, dtype=dtype)) #
            learnable gating parameter - initialize at 0
        self.alpha_dense = th.nn.Parameter(th.tensor(0, dtype=dtype)) #
            learnable gating parameter - initialize at 0
```

```
        self.dropout = th.nn.Dropout(p=0.2, inplace=False)

    def forward(
        self,
        x, # input - keys
        y, # input -  QUERY - input shape of this is output shape of cross-attention
        attn_mask = None
        ):

        """Applies a GATED XATTN-DENSE layer."""

        # 1. Gated Cross Attention
        cross_attention = self.attention(query=y, key=x, value=x, attn_mask=attn_mask)[0]
        cross_attention = self.dropout(cross_attention)
        c = y + (cross_attention * th.tanh(self.alpha_xattn))

        #print("GXA output isNaN:",th.isnan(cross_attention).any())

        # 2. Gated Feed Forward (dense) Layer
        ffw = self.ffw(c)
        ffw = self.dropout(ffw)
        o = c + (ffw * th.tanh(self.alpha_dense))

        return o # output

# define cross atention layers from VPT->LM, LM->VPT
self.Xattn_VPT_LM = MaskedGatedCrossAttention(
    embed_dim=self.LM.transformer.d_embed, # queries are lanuage tokens - LM gets appropriate size and number of tokens
    kvdim=hidsize, # embedding dimensions for VPT transformer layers
    num_heads=attention_heads,
    ffw_dim=hidsize*pointwise_ratio, # FFW hidden size should be same architecturally as FFW in LM (i.e. ratio between transformer token size and FFW hidden layer should be same) (x2 to account for multimodal tokens)
    batch_first=True,
    dtype=self.dtype)
```

5.3.3 The IDM inference code

This is the code used to label video with actions from the Inverse Dynamics Model (IDM) pretrained in the VPT paper.

```
c
```

5.3.4 The data loader

```
d
```

5.3.5 Transformer XL Language model batch order/memory manipulation. Similar code is also used for saving/loading hidden states for the VPT model and recurrent cross attention

In order to keep homogeneity of data during training to avodi spurious correlations, the VPT paper shuffles the data such that we are sampling from many different viddeos durign training depsite only having a small batch size - for each batch a random set of teh video are chosen to progress adn train on. Since the VPT model is recurrent, this requires lodaing adn storing the appropriat hidden state/'memories'. While this is done easily with a matrix called 'first' in the code, teh transformerXL langauge model provides no way to do this, so it is added manually by creating a custom function to save, manipulate and load the appropriate hiddens states according to which videos appear in that batch. For customizeability and for the sake of removing a memory leak in the original VPT code, the VPT model is also given this sort of memory saver/loader instead of its regular one. Since the recurrent cross attention also has persistent memories, a loader is ritten for thsi as well. For brevity, neither of these are shown. This code is in the 3 main training programs. @

```
1
2  def load_hidden_states_LM(video_ids, saved_hidden_states):
3      assert isinstance(video_ids, list)
4      try:
5        T = policy.net.LM.transformer.mem_len
6        n_layers = policy.net.LM.transformer.n_layer
7      except:
8        return None
9      B = BATCH_SIZE
10     E = 1024
11
12     out_hidden_state = []
13     for i in range(n_layers):
14         out_hidden_state.append(th.zeros([T,B,E], dtype=DTYPE).to(DEVICE))
15
16     for b, video in enumerate(video_ids):
17
18         if video in saved_hidden_states:
19             hidden_state = saved_hidden_states[video]
20         else:
21             hidden_state = policy.net.LM.transformer.init_mems(1)
22
23         for l in range(n_layers):
24             out_hidden_state[l][:T,b,:E] = hidden_state[l].clone().squeeze
                   (1)
25
26
27     return out_hidden_state
28
29 def save_hidden_states_LM(video_ids, hidden_state, saved_hidden_states):
30     try:
31       T = policy.net.LM.transformer.mem_len
32       n_layers = policy.net.LM.transformer.n_layer
33     except:
34       return None
35     B = BATCH_SIZE
```

```
    E = 1024

    for b, video in enumerate(video_ids):
        out_hidden_state = []
        for layer in hidden_state: # rewrite with the new one
            layer_sample = layer[:T,b,:E].clone().unsqueeze(1)
            out_hidden_state.append(layer_sample)

        saved_hidden_states[video] = out_hidden_state
```

5.3.6 Training a classifier to identify clean Minecraft videos automatically

This was unused in the end when it was realised that the target dataset size was small enough that manually labelling was feasible and would result in cleaner data

```

# LOAD DATASET

with open('spamham_samples/frames_labels.txt') as file:
    frames_labels = file.readlines()
    random.shuffle(frames_labels)


# load labels. ONLY LOAD 1 FRAME PER VIDEO
links = []
labels = []
for i in range(len(frames_labels)):
    link = frames_labels[i].split(',')[0]
    if not link in links:
        links.append(link)
        label = frames_labels[i].split(',')[1]
        labels.append(label)

num_samples = len(frames_labels)

# LOAD & EMBED IMAGES WITH CLIP EMBEDDING
# from images, create image features from CLIP
x_image_features = []
y_labels = []
missing=0
for i in range(num_samples):
    try:
        image_name = 'spamham_samples/frames/' + ''.join(frames_labels[i].
            split(',')[0:-1]) + '.jpeg'
        with Image.open(image_name) as image:
            image = preprocess(image).unsqueeze(0).to(device)
        with th.no_grad():
            x_image_features.append(model.encode_image(image))
        y_labels.append(labels[i])

        #display progress
        if i\%(num_samples//100)==0:
            print('=',end='')
    except:
        missing+=1
        pass
```

```
x_image_features = th.cat(x_image_features)
print('missing',missing)
num_samples -= missing
print(num_samples)
```

5.3.7 Code for transcribing the audio during a video and converting it to the language model's tokenizer while maintaining word-level timestamps

```
# from stable whisper, extract words and timestamps
print('transcribing', filename)
results = model.transcribe(input_videos_location+filename)#, language='en'
stab_segments = results['segments']
first_segment_word_timestamps = stab_segments[0]['whole_word_timestamps']
stab_segments = stabilize_timestamps(results, top_focus=True)

ASR_words=[]
ASR_secs=[]
for i in range(len(stab_segments)):
    for j in range(len(stab_segments[i]['word_timestamps'])):
        wordms = stab_segments[i]['word_timestamps'][j]
        token = wordms['word']
        ms = wordms['timestamp']
        #print(token, ms)
        ASR_words.append(token)
        ASR_secs.append(ms)









# convert words to agent's LM tokens and transfer ms timestamps accordingly - HARD. NOTE: ASSUMES THAT decode(ASR_tokenizer(text)) = decode(GPT2_tokenizer(text)), WHICH MAY NOT BE TRUE
# NOTE: The Transfo_XL LM used has not BOS token. This code keeps this in mind.
string = ''.join(ASR_words)
agent_words_index = tokenizer(string)['input_ids'] #remove EOS token
# decode token indices to strings
agent_words=[]
for t in range(len(agent_words_index)):
    agent_words.append(tokenizer.decode([agent_words_index[t]]))

word_ms = [0]*len(agent_words)
tok1_index,ch1_index = 0,0
tok2_index,ch2_index = 0,0
current_token_timestamps = [] # is modified though. whe n a new agent token is found is is rest to empty. wesearch through every timestamp that token could occur at and save it to list list. we then decide when the token occured.
step=0
while tok2_index<len(agent_words) and tok1_index<len(ASR_words): # keep going until all agent words have timestamps

```

```
    # according to timestamp at current ASR token that the curretnASR character index is at, give the timestamp associated with that word to the current token that agents character index is at.
    current_token_timestamps.append(ASR_secs[tok1_index])

    char1 = ASR_words[tok1_index][ch1_index] #['hi ', ' there', ' how']
    char2 = agent_words[tok2_index][ch2_index] #['hi ', ' there', ' how']
           print(step)

    print(ASR_words[tok1_index], char1, char2, tok1_index, tok2_index)


    # ensure that we are at the same point in speech in both texts. If not, could be ebcause ok tokenizer differences: accounted for here: 1) ASR uses spaces in outputted words, agent tokenizer does not. 2) agent tokenizer outputs <UNK> for unknown words.
    if char1 != char2:
        print('mismatch on: "'+char1+'", "'+char2+'"')
        if char1==' ': #deak with ASR tokeniser giving spaces berfore/after words and agent tokenizer not doing so
            ch1_index+=1 # ASR words have spaces, agent does not. so we ignore space difference and scooth char1 ahead if it hits a space. now that we have done this change, restart the iteration so we can check if they are now different again or need to be shifted again because 'Hello ', ' there'
            if ch1_index == len(ASR_words[tok1_index]):
                print('1')
                tok1_index += 1
                ch1_index=0
            continue
        if char2 == '<': # deal with <UNK> - add it and timestamp and skip past rest of <UNK> characters
            final_timestamp = min(current_token_timestamps)
            #if word_ms[tok2_index-1] == final_timestamp:
            #    final_timestamp += 1 # it doesnt actually matter if one token in ASR ends up as two tokens in agent_words, as long as order is preserved, since the agent formats them into one token per frame and uses D, so it takes close otegther words back roughly into average WPM anyway. What this does is if one ASR token produces two Agent tokens, they will have the same timestamp, so we need to say the second one happened 1ms later than it did to preserve order.
            word_ms[tok2_index] = final_timestamp # save agent token ms to this list. This and agent_words
            tok2_index += 1
            ch2_index = 0
            current_token_timestamps=[]
            continue
        elif char2 == '@':   # agent tokenizer encodes '2.0' to ['2','@.@','0'] :(
            final_timestamp = min(current_token_timestamps)
            #if word_ms[tok2_index-1] == final_timestamp:
            #    final_timestamp += 1 # it doesnt actually matter if one token in ASR ends up as two tokens in agent_words, as long as order is preserved, since the agent formats them into one token per frame and uses D, so it takes close otegther words back roughly into average WPM anyway. What this does is if one ASR token produces two Agent tokens, they will have the same timestamp, so we need to say the second one
```

```
                happened 1ms later than it did to preserve order.
72              word_ms[tok2_index] = final_timestamp # save agent token ms to
                    this list. This and agent_words
73              tok2_index += 1
74              ch2_index = 0
75              current_token_timestamps=[]
76              continue
77          else:
78              # deal with <UNK>
79              ch1_index+=1 # ASR words have spaces, agent does not. so we
                    ignore space difference and scooth char1 ahead if it hits a
                     space. now that we have done this change, restart the
                    iteration so we can check if they are now different again
                    or need to be shifted again because 'Hello ', ' there'
80              if ch1_index == len(ASR_words[tok1_index]):
81                  print('1')
82                  tok1_index += 1
83                  ch1_index=0
84              continue
85              #assert (char1 == char2), "tokenizer differences: original
                    string cannot be converted"
86
87      # move CHAR1 up one
88      if ch1_index == len(ASR_words[tok1_index])-1:
89          tok1_index += 1
90          ch1_index=0
91      else:
92          ch1_index+=1
93
94      # move CHAR2 up one
95      if ch2_index == len(agent_words[tok2_index])-1:
96          # reached end of current token. We have all ms that characters in
                token occurred across. just assume that this whole token
                occurred at first character's ms
97          # final timestamp is just first observed subtimestamp, so that
                frame and word starts are aligned. This is important becasue it
                 retain causality. The point is also to get across linguitic
                information, not copy timing, and when a word is stated is when
                 the information is  decided, so that is what should be used to
                 most acurately most linguistic information across time
98          final_timestamp = min(current_token_timestamps)
99          #if word_ms[tok2_index-1] == final_timestamp:
100         #    final_timestamp += 1 # it doesnt actually matter if one token
                in ASR ends up as two tokens in agent_words, as long as order
                is preserved, since the agent formats them into one token per
                frame and uses D, so it takes close otegther words back roughly
                 into average WPM anyway. What this does is if one ASR token
                produces two Agent tokens, they will have the same timestamp,
                so we need to say the second one happened 1ms later than it did
                 to preserve order.
101
102         word_ms[tok2_index] = final_timestamp # save agent token ms to this
                 list. This and agent_words
103         tok2_index += 1
104         ch2_index=0
105
106         current_token_timestamps = [] # reset timestamps associated with
                current token for next token
107     else:
```

```
        ch2_index+=1


    step += 1




# get final [words, ms] an save to file
import numpy as np
final_agent_words=agent_words_index
final_agent_words_ms = (np.asarray(ASR_secs)*1000).astype(np.uint32)

output_filename = output_location+link

with open(output_filename,'w') as file:
    pass

with open(output_filename, 'a') as file:
    for i in range(len(final_agent_words)):
        line = str(final_agent_words[i]) + ',' + str(final_agent_words_ms[i
            ]) + '\n'
        file.write(line)
```

In order to calrify which files are contributed by this work, as opposed to which are from the original VPT code provided, we make a list here of files in teh code, which have been modified and brief descriptions of their purpose

```
_DISSERTATION

    - foundation-model-1x.model.model {from VPT repo.} [the model file for
        the VPT agent used]
    - foundation-model-1x.model.weights {from VPT repo} [the pretrained
        weights for the VPT agent used]

    -DATASET {a novel dataset} [a folder containing the collected dataset,
        using methods in the Data Gathering folder]

        -train [a folder containing training data]
            -videos [contains the original .mp4 video files from YouTube]
                ...mp4
            -numpy256 [contains the pre-processed video frames]
                ...npy
            -actions [contains the estimated action labels from the IDM]
                ...IDMaction
            -transcipts [contains the tokenised transcripts for the videos
                with per-token timestamps]

        -valid [a folder containing evaluation data]
            ... (same as train ^)

    Data Gathering {novel code, ideas from VPT paper} [The collection
        method is followed from the VPT paper but all code is written by
        this work]



    -VLPT
```

```
        - 4x_idm.model {from VPT} [the IDM model]

        - 4x_idm.weights  {from VPT} [the IDM model]

        - agent.py {mildly modified from VPT} [this creates a MineRLAgent
            wrapper around the pytorch model policy. Converting words from
            a string to silence-token-padded tokens is added, support for
            language input and output is added including autoregression
            during mineRL inference, special inference modifications for
            providing language instructions to the agent during inference
            in MineRL is provided, as well as printing the models outputted
             reposnses.]

        - BLC_evaluate.py {mostly novel, adapted from behavioural_cloning.
            py in VPT} [converts the basic training pipeline into an
            evaluation one]

        - IDM_data_loader_np256.py {strongly modified} [this has taken the
            original threaded dataloader from VPT, but modified to enable
            loading actions predicted from the IDM rather than .jsons of
            human behaviour; to enable loading the transcriptions and
            syncronize them with actions and video, including adding
            silence tokens; to shuffle when episodes start to homogenise
            episode restarts during training; to load video from numpy
            files for much improved data loading speed]

        - IDM_labeller.py {mostly novel} [this was originally
            run_inverse_dynamics_model.py in VPT, which took in an mp4 and
            a .json file of correct action labels, and display the video
            and the IDMs estimated actions against the correct actions,
            convolving the IDm model across the video with no overlaps. It
            has been modified to run on minecraft videos with no .jsons and
             produce estimated actions files and save them. It is now
            convolved with overlaps such as only the middle 64 action
            estimations are used as opposed to the full 128 that it
            estimates in order to reduce causal inference and improve
            performance, as is done with teh IDM model in teh VPT paper.
            Batching is also enabled, and multithreading is enabled to
            allow for faster video loading and inference. Some
            preprocessing steps have been added to allow videos with black
            borders to be processed]

        - inverse_dynamics_model.py {mostly VPT} [somewhat modified to
            allow the IDM to save actions in a more useful format]


        -lib {FOLDER}
           - action_head.py {from VPT}
           - action_mapping.py {from VPT}
           - actions.py {from VPT}
           - data_parallel.py {from VPT}

           - IDM_policy.py {from VPT} [separated out from the agent policy
               by myself.]

           - impala_cnn.py {from VPT, modified by myself to support dropout
                and bfloat16}

           - __init__.py {from VPT}
```

```
51            - masked_attention.py {from VPT}
52
53            - masked_gated_cross_attention.py {mostly my own} [this file was
                  added by myself and takes inpiration from lucidsrains, 2022
                  implementation of the Flamingo gated cross attention. It
                 imports some modules from the VPT repo.]
54
55            - minecraft_util.py {from VPT}
56            - misc.py {from VPT}
57            - mlp.py {from VPT}
58            - normalize_ewma.py {from VPT}
59            - scaled_mse_head.py {from VPT}
60            - torch_util.py {from VPT}
61            - tree_util.py {from VPT}
62
63            - util.py {from VPT, modified to support dropout and bfloat16}
64
65            - VLPT_policy.py {mix} [this is where the key architecture is
                implemented - the policy definition for the agent. Most code
                 is maintained from the VPT implementation, but much is
                removed for simplicity. The core VPT model definition
                remains, but has much added onto it - support for dynamic
                causal mask creation in the forward method for Xattn and the
                 LM, gated cross attention, a fused language model, bloat16
                support, special treatement of inference code to
                accomodation LM timeout rate, and all of the re-writing of
                how data is passed between classes in order to accomodate
                for these extra inputs and parameters (i.e. language or
                xattn related) is done by myself.]
66
67            - xf.py {from VPT}
68
69        - LICENSE - VPT {from VPT}
70
71        - modeling_transfo_xl_16.py     {the following few files are from
              the Transformer-XL source code (Yang et al, 2019), specifically
               the HuggingFace (HuggingFace, 2019) implementation and are
              modified to allow for more efficient loss prediction (1 token
              is usually discarded) and retreiving the loss as well as the
              predicted words in a single call.}
72        - modeling_transfo_xl.py
73        - modeling_transfo_xl_utilities_16.py
74        - modeling_transfo_xl_utilities.py
75        - transformerXL_LICENSE
76
77        - README - VPT.md {from VPT}
78
79        - requirements_linux.py {mostly novel} [installs most required
              dependencies for the project]
80
81        - run_agent.py {strongly modified} []
82
83        - TRAIN_behavioural_cloning.py {mostly novel} [modified from
              behavioural_cloning.py. most things have been re-implmenented
              in a custom way for the need of this project and much has been
              added in way of evaluating on the validation set, logging
              multiple graphs, adding dupport for language input and tracking
               hidden states for VPT, LM and X-attn. This is one of the key
              contributions. It has been made much more efficient and has had
```

```
            memory leaks removed and has been tried and tested in a real
            cloud training envronment, and is not just an example/test
            script anymore.]
        - TRAIN_behavioural_language_cloning.py {same ^}
        - TRAIN_language_cloning.py {same ^}
        - BLC_evaluation.py {same ^}

        - video_numpier2.py {novel} [converts .mp4 videos into .npy chunks
            for faster loading during training by the data loader]

    -TRAINING
        -LM_ONLY {novel} [contains many folders with logs, graphs and
            weights from many different training runs of language model
            training]

        -VPT_ONLY {novel} [contains many folders with logs, graphs and
            weights from many different training runs of the VPT agent
            training]

        -MIXED {novel} [contains folders with logs and weights fromtraining
             runs of joint training attempts]
```